Efficient and Robust Allocation Algorithms in Clouds under Memory Constraints

Olivier Beaumont
Inria Bordeaux, France
Olivier.Beaumont@inria.fr

Lionel Eyraud-Dubois
Inria Bordeaux, France
Lionel.Eyraud-Dubois@inria.fr

Juan-Angel Lorenzo
Inria Bordeaux, France
Juan-Angel.Lorenzo-del-Castillo@inria.fr

Paul Renaud-Goud
Inria Bordeaux, France
Paul.Renaud-Goud@inria.fr

ABSTRACT

We consider robust resource allocation of services in Clouds. More specifically, we consider the case of a large public or private Cloud platform such that a relatively small set of large and independent services accounts for most of the overall CPU usage of the platform. We will show, using a recent trace from Google, that this assumption is very reasonable in practice. The objective is to provide an allocation of the services onto the machines of the platform, using replication in order to be resilient to machine failures. The services are characterized by their demand along several dimensions (CPU, memory,...) and by their quality of service requirements, that have been defined through an SLA in the case of a public Cloud or fixed by the administrator in the case of a private Cloud. This quality of service defines the required robustness of the service, by setting an upper limit on the probability that the provider fails to allocate the required quantity of resources. This maximum probability of failure can be transparently turned into a set of $(price, penalty)$ pairs.

Our contribution is two-fold. First, we propose a formal model for this allocation problem, and we justify our assumptions based on an analysis of a publicly available cluster usage trace from Google. Second, we propose a resource allocation strategy whose complexity is low in the number of resources, what makes it well suited to large platforms. Finally, we provide an analysis of the proposed strategy through an extensive set of simulations, showing that it can be succesfully applied in the context of the Google trace.

Categories and Subject Descriptors

F.2.0 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*General*

Keywords

Cloud Computing; resource allocation; reliability; column generation; probability estimation; replication; resilience

1. INTRODUCTION & RELATED WORKS

Recently, there has been a dramatic change in both the platforms and the applications used in parallel processing. On the one hand, we have witnessed an important scale change, that is expected to continue both in data centers and in exascale machines. On the other hand, an important simplification change has also occurred in the application models and scheduling algorithms. On the application side, many large scale applications are expressed as (sequences of) independent tasks, such as MapReduce applications [1, 2] or even run as independent services handling requests, as in the case of a PaaS (Platform as a Service) Cloud.

In fact, the main reason behind this paradigm shift is not related to scale but rather to unpredictability. First of all, estimating the duration of a task or the time of a data transfer is extremely difficult, because of NUMA effects, shared platforms, complicating network topologies and the number of concurrent computations/transfers. Moreover, given the number of involved resources, failures are expected to happen at a frequency such that robustness to failures is a crucial issue for large scale applications running on Cloud platforms. In this context, the cost of purely runtime solutions, agnostic to the application and based either on checkpointing strategies [3, 4] or application replication [5] is expected to be unnecessarily large in the context of regular applications, and there is a clear interest for application-level solutions that take the inner structure of the application to enforce fault-tolerance.

In this paper, we will consider the context of a PaaS Cloud (Platform as a Service), in which several independent services are handling queries and need to be allocated onto physical machines (PMs) [6, 7]. In the static case, mapping services with heterogeneous computing demands onto PMs with capacities is amenable to a multi-dimensional bin-packing problem (each dimension corresponding to a different kind of resource, memory, CPU, disk, bandwidth,...). Indeed, in this context, on the Cloud administrator side, each physical machine comes with its computing capacity

(*i.e.* the number of flops it can process during one time-unit), its memory capacity and its failure rate (*i.e.* the probability that the machine will fail during the next time period). On the client side, each service comes with its requirement along the same dimensions (memory and CPU footprints) and a reliability requirement that has been negociated through an SLA [8].

In order to deal with resource allocation problems in IaaS (Infrastructure as a Service) Clouds, several sophisticated techniques have been developed in order to optimally allocate VMs (Virtual Machines) onto PMs, either to achieve good load-balancing [9, 10] or to minimize energy consumption [11]. Most of the works in this domain are based on offline [12] and online [13] solutions of Bin Packing variants. However, reliability constraints have received much less attention in the context of Cloud computing, as underlined by Cirne *et al.* [8].

In this paper, we will consider reliability issues in a simplified context, although representative of many Cloud platforms, as we will demonstrate it in Section 2.2 using a publicly available trace from a Google cluster [14]. We assume that each service represents a divisible workload, in the sense that it is possible to allocate its workload freely upon the machines of the platform. This is indeed the case for many Internet services (like merchant websites for example), for which it is possible to have a load-balancing frontend which forwards the requests in the correct proportions to each virtual machine. In a fashion similar to a previous work [8], we consider reliability in a static scenario, and our objective is to compute allocations that are resilient to failures thanks to overprovisioning: each service is allocated more resources than needed in order to enforce that enough resource is available to serve requests if failures happen. More specifically, our goal is to find an allocation that will remain valid with a given probability in presence of failures (described through a probability distribution).

This rather coarse-grain model (where we consider services instead of individual virtual machines) makes it possible to propose sophisticated allocation algorithms (based on powerful techniques such as column generation) while considering platforms with relatively large number of machines. Indeed, as shown in Section 2.2, a rather small fraction of all long running services consists of many tasks and represents a majority of the resource usage. It is thus meaningful to concentrate the computational efforts for the allocation on those large services. Moreover, in the trace, a large fraction of these long running jobs belongs to high priority classes, thus justifying the importance of reliability requirements.

This work is a follow-up of [15] and [16], where reliability of service allocation problems have been considered, but in the context where services are defined by their processing requirement only (and not their memory requirement) and in the context of IaaS Clouds. One of the main results of [15] is that estimating the reliability of a given allocation is already a #P-complete problem [17]. In [16], energy minimization is considered, and asymptotically optimal approximation algorithms are proposed, based on the use of Chernoff and Hoeffding bounds. In the present paper, we improve on this previous work on several fronts:

- We have enhanced the realism of the model by adding memory requirements for services, which we tackle with an algorithm that globally optimizes the number of machines each services is alloted to, given the available resources;
- We prove that our model is compliant with the characteristics of a class of services (jobs in [18] terminology) of the previously mentioned Google trace data [14]. This set of services (each service consisting in many long running tasks) is relatively small (so that sophisticated optimization techniques can be used on it) but still encompasses most of the CPU usage of the platform.
- We propose a more powerful packing heuristic to perform the actual allocation of services, based on column generation techniques (see Section 3.2);

The paper is organized as follows. In Section 2, we present the notations that will be used throughout this paper and we define the characteristics of both the platform and the services that are suitable for the techniques we propose. Our assumptions are justified by an analysis of a cluster usage trace made available by Google [14]. In Section 3, we propose an algorithm for solving the resource allocation problem under reliability constraints. It relies on a pre-processing phase that is used to decompose the problem into a reliability problem and a packing problem. Finally, we present in Section 4 a set of detailed simulation results that enables us to analyze the performance of the algorithm proposed in Section 3 both in terms of the quality of returned allocations and processing time. Concluding remarks are presented in Section 5.

2. FRAMEWORK

2.1 Platform and services description

In this paper, we assume the following model. On the one hand, the platform is composed of m homogeneous machines $\mathcal{M}_1, \ldots, \mathcal{M}_m$, that have the same CPU capacity C and the same memory capacity M. Machine heterogeneity will be discussed in the next section. On the other hand, we aim at running ns services $\mathcal{S}_1, \ldots, \mathcal{S}_{ns}$, that come with their CPU and memory requirements. As stated in the introduction, we assume that the CPU demand of a service can be spread across several machines, with a fluid CPU sharing model: on a given machine, the fraction of the total CPU dedicated to a given service can take any (rational) value. This expresses two assumptions: (i) the sharing between services running on a given machine is done through time multiplexing, whose grain is very fine, and (ii) the workload of each service can be balanced among all its running instances, for example with a frontend which dynamically forwards the requests to the appropriate instance.

However, we consider that the memory requirements of a service represent the amount of memory used by any instance of this service, regardless of the amount of computation power allocated to this service on this machine. This assumption models the fact that most of the memory used by virtual machines comes from the complete software stack image that needs to be deployed, and is compliant with our observations from the Google trace data in the next section.

In this context, our goal is to minimize the number of used machines, and to find an allocation of the services onto the machines such that all packing constraints are fulfilled, and such that the resource requirements of the service are met in a *robust* manner, even in the presence of machine failures.

2.2 Model discussion from trace analysis

Google recently released a complete usage trace from one of its production clusters [14]. Several teams have analyzed this trace to describe the associated workload [19]. We tested some of the assumptions of our model on this trace, and we present the conclusions in this section.

Data from Google shows that heterogeneity is rather low in this cluster: a large group (more than 50%) of the machines is homogeneous and, with four homogeneous classes, it is possible to cover 98% of the machines.

The workload in the Google trace consists in many services (they are called *jobs* in the terminology of this trace), further divided into *tasks*, each task being run onto a single machine. Among other things, the trace provides the *priority* class of each service, and reports the memory and cpu usage of each task on 5-minute intervals. The whole trace spans a time of about a month.

The first observation that can be made with this data is that most of the workload (in terms of cpu usage) comes from a relatively small set of services. Indeed, it can be observed that at any time in the trace, the largest 150 services account for at least 90% of the overall CPU usage and that the largest 500 services account for 99% of the CPU usage. Therefore, even though the overall trace involves a huge number of services, it is of interest to concentrate on a small fraction of them (100 to 500 services) that accounts for most of the platform usage. This justifies our assumption that it is possible to focus the optimization on relatively few services, and thus use sophisticated allocation techniques. Note that most of these services represent a large number of tasks, what explains their overall weight. It is therefore crucial for the algorithms to concentrate on services (or jobs) and not on tasks (or virtual machines).

The context of this paper is also based on the fact that services are user-facing jobs that run for very long durations and require strong reliability guarantees. By examining the trace, it is possible to identify the services which exhibit these characteristics. Indeed, one of the higher priority classes is the *production* class, so we have selected services from this class. To identify long running jobs, we have picked at random 30 5-minute intervals in each day of the trace, and we have isolated the services which are active on at least 90% of these intervals (in order to cope with missing data). This yields a rather large number of services (1240), but most of them use a very small amount of CPU. By removing the services which use less than the capacity of a typical machine of the cluster, we obtain about 120 services (between 90 and 150 depending on the time interval) which account for 96% of the CPU usage of the original 1240 services. Those large, high-priority, long-running services account for a significant part of the total CPU usage of the cluster: on average, about 35%.

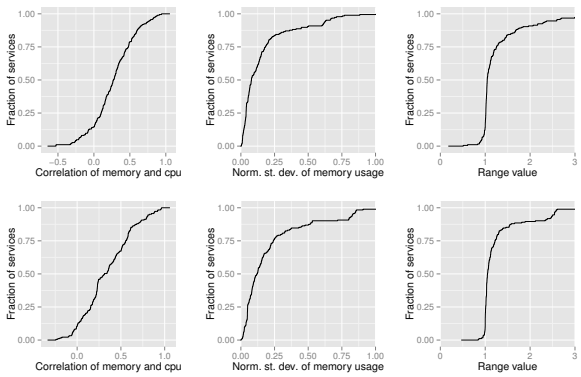

Figure 2: Analysis of the dependency between memory usage and cpu usage for the tasks of services in two different datasets: Large on the top, HpLr on the bottom.

Another assumption of our model states that memory usage for a service on a machine does not depend on the amount of cpu allocated to this service on that machine. To test this assumption, we have analyzed two sets of jobs: (i) the services that account for 90% of the CPU usage on 20 different snapshots all over the duration of the trace, and that form the Large dataset and (ii) the high-priority, long running services described above that form the HpLr dataset. In both cases, for each service, we have analyzed simultaneously the memory and cpu usage of each of their individual tasks, and we have observed that for most of the services, the memory usage of a task is almost independent of the cpu usage of that task. This can be seen in Figure 2, by three different criteria:

- The correlation factor between cpu usage and memory usage is rather low: the absolute value of the correlation is below 0.5 for 77% of the Large services and 67% of the HpLr services, and it is below 0.4 for 66% of the Large services and 58% of the HpLr services.
- The variance of the memory usage for a given service is rather low: the relative standard deviation (the standard deviation normalized by the mean value) is below 0.25 for 83% of the Large services, and for 77% of the HpLr services.
- For each service, we have performed linear regression to analyze the dependency of memory usage on cpu usage, and the resulting slopes are small for most of the services. To account for scaling issues (the values of the slope depends on the units used to express cpu and memory usage), we have computed the *range*, defined as the ratio between the value estimated by the linear regression for the median cpu usage and for 0 cpu usage. This range is below 1.25 for 75% of the Large services and 77% of the HpLr services, and below 1.5 for 84% of the services in both datasets.

If we consider the services which show low dependency for all three criteria (correlation below 0.4, relative standard

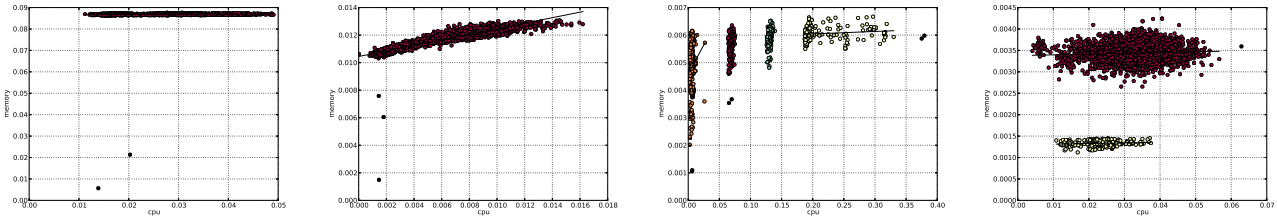

Figure 1: Dependency of memory usage vs cpu consumption of four different services. The leftmost plot shows no dependency between memory and cpu. The other three plots show counter-examples.

deviation below 0.25 and range below 1.25), we get 52% of the HPLR services, which account for 60% of their CPU usage, and in the case of the LARGE services, they represent 59% of the services and 65% of the CPU usage.

These findings show that for a large proportion of the services, the memory usage of a task does not depend on its cpu usage, what validates our model. Typical situations for (large) services are depicted in Figure 1. In Figure 1, each plot corresponds to a service and each point corresponds to the (memory,cpu) profile of one task of this service at a given time stamp. The leftmost plot corresponds to the case where the memory usage is independent of the CPU usage, what is the most classical situation as shown by statistical distributions depicted in Figure 2. We can however observe a few tasks that can be considered as outliers as shown in the 3 other plots. The second plot shows a small but significant correlation between CPU and memory usages. The third service cannot easily be incorporated into our model. Indeed, it exhibits several types of tasks (corresponding to the vertical bars) and for each kind of task, there is a large discrepancy in the memory usage. Incorporating such services into our model is left for future work, but such services are highly untypical in the trace. At last, the rightmost picture exhibits two classes of tasks (those with memory usage of order 3.5×10^{-3} and those with memory usage of order 1.3×10^{-3}). In order to be compliant with our model, such a service needs to be split into two different services. Again, doing this operation automatically on all services is left for future work.

2.3 Failure model

In this paper, we envision large-scale platforms, what means that machine failures are not uncommon and need to be handled explicitly. Two techniques are usually set up to face up machine failures: migration and/or replication. Following [8], we consider that the response time of migrations is too high to ensure continuity of the services and we therefore concentrate on the use of overprovisioning in order to provide resilience with very quick recovery time. This overprovisioning allows a service to continue running seamlessly after possible failures and the system can replace failed machines (using a set of backup resources) to obtain the same reliability guarantees.

More specifically, we assume that machine failures are independent, and that machines are homogeneous also with regard to failures. We denote f the probability that a given machine fails during the time period between two migration phases. Because of those failures, we cannot ensure that a service will have enough computational power at its disposal during the whole time period. The probability that all machines in the platform fail is indeed positive. Therefore, in our model each service $\mathcal{S}_i$ is also described with its reliability requirement r_i, which expresses a constraint: the probability that the service has not enough computational power (less than its demand d_i) at the end of the time period must be lower than r_i.

In this context, by replicating a given service on many machines whose failure are independent, it will be possible to achieve any reliability requirement. Our goal in this paper is to do it for all services simultaneously, *i.e.* to enforce that capacity constraints, reliability requirements and service demands will be satisfied, while minimizing the number of required machines.

2.4 Problem description

We are now ready to state precisely the problem. We have m homogeneous machines with CPU capacity C, memory capacity M, and failure probability f. We also have ns services, where service $\mathcal{S}_i$ has a CPU requirement d_i, memory requirement m_i, and a reliability constraint r_i. An *allocation* is defined by $A_{i,j}$, the CPU allocated to service $\mathcal{S}_i$ on machine $\mathcal{M}_j$, for all $i \in \{1, \ldots, ns\}$ and $j \in \{1, \ldots, m\}$. For all $j \in \{1, \ldots, m\}$, we denote is_alive_j the random variable which is equal to 1 if machine $\mathcal{M}_j$ is alive at the end of the time period, and 0 otherwise. We can then define, for all $i \in \{1, \ldots, ns\}$, the total CPU amount that is available to service $\mathcal{S}_i$ at the end of the time period: $Alive\_cpu_i = \sum_{j=1}^{m} is\_alive_j \times A_{i,j}$. Furthermore, we use the notation $\mathbb{1}_{A_{i,j}}$, which is equal to 1 if $A_{i,j}$ is positive, and 0 otherwise. The problem of the minimization of the number of used machines can be written as follows, where equations (2) and (3) depict the packing constraints, while Equation (1) deals with reliability requirements:

$$\min_{m} \text{ s.t. } \begin{cases} \forall i, \mathbb{P}\left(Alive\_cpu_i < d_i\right) < r_i & (1) \\ \forall j, \sum_{i=1}^{ns} m_i \mathbb{1}_{A_{i,j}>0} \leq M & (2) \\ \forall j, \sum_{i=1}^{ns} A_{i,j} \leq C & (3) \end{cases}$$

We will use in this paper two approaches for the estimation of the reliability requirements. In the No-Approx model, the reliability constraint is actually written $\mathbb{P}(Alive\_cpu_i < d_i) < r_i$.

However, as previously stated, given an allocation of one service onto the machines, deciding whether this allocation fulfills the reliability constraint or not is a #P-complete problem [15]; this shows that estimating this reliability constraint is a hard task. In [20], it has been observed that, based on the approximation of a binomial distribution by a Gaussian distribution, $\mathbb{P}(Alive\_cpu_i < d_i) < r_i$ is approximately equivalent to

$$\sum_{j=1}^{m} A_{i,j} - B_i \sqrt{\sum_{j=1}^{m} A_{i,j}^2} \geq K_i, \quad \text{where}$$

$$K_i = \frac{d_i}{1-f} \quad \text{and} \quad B_i = z_{r_i} \times \sqrt{\frac{f}{1-f}}.$$

z_{r_i} is a characteristic of normal distributions, and only depends on r_i, therefore it can be tabulated beforehand. In the following, we will denote this model by the Normal-Approx model.

Both packing and fulfilling the reliability constraints are hard problems on their own, and it is even harder to deal with those two issues simultaneously. In the next section, we describe the way we solve the global problem, by decomposing it into two sub-problems that are easier to tackle.

3. PROBLEM RESOLUTION

We approach the problem through a two-step heuristic. The first step focuses mainly on reliability issues. The general idea about reliability is that, for a given service, in order to keep the replication factor low (and thus reduce the total number of machines used), the service has to be divided into small slices and distributed among sufficiently many machines. However, using too many small slices for each service would break the memory constraints (remember that the memory requirement associated to a service is the same whatever the size of slice, provided it is positive). The goal of the first step, described in Section 3.1, is thus to find reasonable slice sizes for each service, by using a relaxed packing formulation which can be solved optimally.

In a second step, described in Section 3.2, we compute the actual packing of those service slices onto the machines. Since the number of different services allocated to each machine is expected to be low (because of the memory constraints), we rely on a formulation of the problem based on the partial enumeration of the possible configurations of machines, and we use column generation techniques [21] to limit the number of different configurations.

3.1 Focus on reliability

In this section, we describe the first step of our approach: how to compute allocations that optimize the compromise between reliability and packing constraints, under both No-Approx and Normal-Approx models. This is done by considering a simpler, relaxed formulation of the problem, that can be solved optimally. We start with the Normal-Approx model.

3.1.1 Normal-Approx *model*

As stated before, in this first phase, we relax the problem by considering global capacities instead of capacities per machine. We thus consider that we have at our disposal a total budget mM for memory requirements and mC for CPU needs, and use the following formulation:

$$\min_{m} \begin{cases} \forall i, \sum_{j=1}^{m} A_{i,j} - B_i \sqrt{\sum_{j=1}^{m} A_{i,j}^2} \geq K_i & (4) \\ \text{s.t.} \quad \sum_{j=1}^{m} \sum_{i=1}^{ns} m_i \mathbb{1}_{A_{i,j}>0} \leq mM & (5) \\ \sum_{j=1}^{m} \sum_{i=1}^{ns} A_{i,j} \leq mC & (6) \end{cases}$$

In the following, we prove that this formulation can be solved optimally. We define a class of solutions, namely *homogeneous* allocations, in which each service is allocated on a set of machines, with the same CPU requirement. Formally, an allocation is *homogeneous* if for all $i \in \{1, \ldots, ns\}$, there exists A_i such that for all $j \in \{1, \ldots, m\}$, either $A_{i,j} = A_i$ or $A_{i,j} = 0$.

Due to lack of space, we refer the reader to the companion research report [22] where all proofs are detailed.

Lemma 1. *On the relaxed problem, homogeneous allocations is a dominant class of solutions.*

An *homogeneous* allocation is defined by n_i, the number of machines hosting service $\mathcal{S}_i$, and A_i, the common CPU consumption of service $\mathcal{S}_i$ on each machine it is allocated to. The problem of finding an optimal *homogeneous* allocation can be written as:

$$\min m \text{ s.t. } \begin{cases} \forall i, n_i A_i - B_i A_i \sqrt{n_i} \geq K_i & (7) \\ \forall i, A_i \geq 0 \\ \sum_i n_i m_i \leq mM \\ \sum_i n_i A_i \leq mC \end{cases}$$

which can be simplified into

$$\min m \text{ s.t. } \begin{cases} \forall i, \sqrt{n_i} > B_i \\ \sum_i m_i n_i \leq mM \\ \sum_i \frac{K_i}{1 - \frac{B_i}{\sqrt{n_i}}} \leq mC \end{cases} \quad (8)$$

In the following, we search for a fractional solution to this problem: m, the n_i's and the A_i's are not restricted to integer values. We begin by formulating two remarks to help solving this problem.

REMARK 1. *We can restrict to solutions for which the two last constraints in Problem* (8) *are equalities.*

REMARK 2. *We define* f_1 *and* f_2 *by* $f_1(n_1,\dots,n_{ns}) = \sum_i m_i n_i$ *and* $f_2(n_1,\dots,n_{ns}) = \sum_i \frac{K_i}{1-\frac{B_i}{\sqrt{n_i}}}$.

Necessarily, at a solution with minimal m*, we have:*

$$\forall i,j \quad \frac{1}{m_i}\frac{\partial f_2}{\partial n_i} = \frac{1}{m_j}\frac{\partial f_2}{\partial n_j}.$$

This shows that for any optimal solution, there exists X such that

$$\forall i \quad -\frac{B_i K_i}{m_i\sqrt{n_i}(\sqrt{n_i}-B_i)^2} = \frac{1}{m_i}\frac{\partial f_2}{\partial n_i} = X.$$

Computing the n_i's given X

By denoting $x_i = \sqrt{n_i}$, let us consider the following third-order equation $x_i(x_i-B_i)^2 + \frac{B_i K_i}{m_i X} = 0$. The derivative is null at $x_i = B_i$ and $x_i = B_i/3$, and the function tends to $+\infty$ when $x_i \to +\infty$. Since we search for $x_i > B_i > 0$, we deduce that for any $X < 0$, this equation has an unique solution. Let us denote $g_i(X)$ the unique value of n_i such that $\sqrt{n_i}$ is a solution to this equation. As $x_i(x_i-B_i)^2 \geq (x_i-B_i)^3$, we know that $x_i \leq B_i + \sqrt[3]{-B_i K_i/m_i X}$. We can thus compute $g_i(X)$ with a binary search inside $]B_i, B_i + \sqrt[3]{-B_i K_i/m_i X}]$, since $x \mapsto x(x-B_i)^2 \leq x^3$ is an increasing function in this interval. Incidentally, we note that for all i, g_i is an increasing function of X.

Computing X

According to remark 2, for any optimal solution there exists X such that

$$\sum_i m_i g_i(X) = M/C \times \sum_i \frac{K_i}{1-\frac{B_i}{\sqrt{g_i(X)}}}. \qquad (9)$$

Since the left-hand side is increasing with X, and the right-hand side is decreasing with X, this equation has an unique solution X^* which can be computed by a binary search on X. Once X^* is known, we can compute the n_i and we are able to derive the A_i's. The solution S^* computed this way is the unique optimal solution: for any optimal solution S', there exists X' which satisfies the previous equation. Since this equation has only one solution, $X' = X^*$ and $S' = S^*$.

We now show how to compute upper and lower bounds for the binary search on X. As shown previously, we have an obvious upper bound: $X < 0$. We express now a lower bound. Let n_i^* be defined, for all i, by

$$m_i n_i^* \geq 0 \ \text{ and } \ m_i n_i^* = \frac{M}{C} \times \frac{K_i}{1-\frac{B_i}{\sqrt{n_i^*}}}.$$

Then $\sqrt{n_i^*}$ is a solution of a second-order equation, $n_i^* - B_i\sqrt{n_i^*} - MK_i/Cm_i = 0$, and since $n_i^* \geq 0$, we can compute:

$$\sqrt{n_i^*} = \frac{1}{2} \times \left(B_i + \sqrt{B_i^2 + \frac{4MK_i}{Cm_i}} \right).$$

Now let

$$X_i = -\frac{B_i K_i}{\sqrt{n_i^*}\left(\sqrt{n_i^*}-B_i\right)^2} \quad \text{and} \quad i^- = \text{argmin}_i\, X_i.$$

For all i, $X_{i^-} \leq X_i$. Since g_i is increasing with X, we have $g_i(X_{i^-}) \leq g_i(X_i) = n_i^*$. Since $x \mapsto K_i/(1-B_i/\sqrt{x})$ is non-increasing, this implies

$$\frac{M}{C}\frac{K_i}{1-B_i/\sqrt{n_i^*}} \leq \frac{M}{C}\frac{K_i}{1-B_i/\sqrt{g_i(X_{i^-})}}.$$

From the definition of n_i^*, we can conclude

$$\sum_i m_i g_i(X_{i^-}) \leq \frac{M}{C} \times \sum_i \frac{K_i}{1-\frac{B_i}{\sqrt{g_i(X_{i^-})}}} \quad \Rightarrow \quad X^* \geq X_{i^-}.$$

With the same line of reasoning, we can refine the upper bound into $X^* \leq X_{i^+}$, where $i^+ = \text{argmax}_i\, X_i$, by showing

$$\sum_i m_i g_i(X_{i^+}) \geq \frac{M}{C} \times \sum_i \frac{K_i}{1-\frac{B_i}{\sqrt{g_i(X_{i^+})}}}.$$

3.1.2 NO-APPROX *model*

In the previous section, we showed how to compute an optimal solution to the relaxed problem under the NORMAL-APPROX model, but we have no guarantee that this solution will meet the reliability constraints under the NO-APPROX model.

Given an homogeneous allocation for a given service $\mathcal{S}_i$, the amount of alive CPU of $\mathcal{S}_i$ follows a binomial law: $Alive\_cpu_i \sim A_i \times \mathcal{B}(n_i, 1-f)$. We can then rewrite the reliability constraint, under the NO-APPROX model, as $\mathbb{P}(A_i \times \mathcal{B}(n_i, 1-f) < d_i) < r_i$. This constraint describes the actual distribution, but since the values (n_i, A_i) have been obtained via an approximation, there is no guarantee that they will satisfy this constraint. However, since the cumulative distribution function of a binomial law can be computed with a good precision very efficiently (see `http://www.gnu.org/software/gsl`), we can compute n_i', the first integer which meets the constraint. We can then use equation (7) to refine the value of B_i: we compute B_i' so that equation (7) with A_i and n_i' is an equality, so that the approximation of the NORMAL-APPROX model is closer to the actual distribution for these given values of A_i and n_i.

We compute new B_i's for all services, and iterate on the resolution of the previous problem, until we reach a convergence point where the values of the B_i do not change. In our simulations (see section 4), this iterative process converges in at most 10 iterations.

3.2 Focus on packing

In the previous section, we have described how to obtain an optimal solution to the relaxed problem 8, in which *homogeneous* solutions are dominant. In the original problem, packing constraints are expressed for each machine individually, and the flexibility of non-homogeneous allocations may make them more efficient. Indeed, an interesting property of equation (4) is that "splitting" a service (*i.e.*, dividing an allocated CPU consumption on several machines instead of one) is always beneficial to the reliability constraint (because splitting keeps the total sum constant, and decreases the sum of squares). In this section, we thus consider the packing part of the problem, and the reliability issues are handled by the following constraints: the allocation of service $\mathcal{S}_i$ on any machine j should not exceed A_i, and the total CPU allocated to $\mathcal{S}_i$ should be at least $n_i A_i$. Since the (n_i, A_i) values are such that the homogeneous allocation satisfies the reliability constraint, the splitting property stated above ensures that any solution of this packing problem satisfies the reliability constraint as well.

The other idea in this section is to make use of the fact that the number k of services which can be hosted on any machine is low. This implies that the number of different machine configurations (defined as the set of services allocated to a machine) is not too high, even if it is of the order of ns^k. We thus formulate the problem in terms of *configurations* instead of specifying the allocation on each individual machines. However, exhaustively considering all possible configurations is only feasible with extremely low values of k (at most 4 or 5). In order to address a larger variety of cases, we use in this section a standard column generation method [21] for bin packing problems.

In this formulation, a configuration $\mathcal{C}_c$ is defined by the fraction $x_{i,c}$ of the maximum capacity A_i devoted to service $\mathcal{S}_i$. According to the constraints stated above, configuration $\mathcal{C}_c$ is *valid* if and only if $\sum_i m_i \lceil x_{i,c} \rceil \leq M$, $\sum_i x_{i,c} A_i \leq C$, and $\forall c, 0 \leq x_{i,c} \leq 1$. Furthermore, we only consider *almost full* configurations, defined as the configurations in which all services except at most one are assigned a capacity either 0 or 1. Formally, we restrict to the set $\mathcal{F}$ of valid configurations $\mathcal{C}_c$ such that $card\,\{i, 0 < x_{i,c} < 1\} \leq 1$.

We now consider the following linear program $\mathcal{P}$, in which there is one variable λ_j for each valid and almost full configuration:

$$\min \sum_{c \in \mathcal{F}} \lambda_c \text{ s.t.} \quad \forall i, \sum_{c \in \mathcal{F}} \lambda_c x_{i,c} \geq n_i \tag{10}$$

Despite the high number of variables in this formulation, its simple structure (and especially the low number of constraints) allows us to use column generation techniques to solve it. The idea is to generate variables only from a small subset $\mathcal{F}'$ of configurations and solve the problem $\mathcal{P}$ on this restricted set of variables. This results in a sub-optimal solution, because there might exist a configuration in $\mathcal{F} \setminus \mathcal{F}'$ whose addition would improve the solution. Such a variable can be found by writing the dual of $\mathcal{P}$ (the variables in this dual are denoted p_i), *i.e.*

$$\max \sum_i n_i p_i \text{ s.t.} \quad \forall c \in \mathcal{F}, \sum_i x_{i,c} p_i \leq 1$$

The sub-optimal solution to $\mathcal{P}$ provides a (possibly infeasible) solution p_i^* to this dual problem. Finding an improving configuration is equivalent to finding a violated constraint, *i.e.* a valid configuration $\mathcal{C}_j$ such that $\sum_i x_{i,c} p_i^* > 1$. We can thus look for the configuration $\mathcal{C}_j$ which maximizes $\sum_i x_{i,c} p_i^*$. This sub-problem is a knapsack problem, in which at most one item can be split.

Let us denote this knapsack sub-problem as Split-Knapsack. It can be formulated as follows: given a set of item sizes s_i, item memory requirements m_i, item profits p_i, a maximum capacity C and a maximum memory size M, find a subset J of items with weights x_i such that $\sum_{i \in J} m_i \leq M$, and $\sum_{i \in J} x_i s_i \leq C$ which maximizes the profit $\sum_{i \in J} x_i p_i$. We first remark that solutions with at most one split item are dominant for Split-Knapsack (which justifies that we only consider almost full valid configurations, *i.e.* configurations with at most one split item). Then, we prove that this problem is NP-complete, and we propose a pseudo-polynomial dynamic programming algorithm to solve it. This algorithm can thus be used to find which configuration to add to a partial solution of $\mathcal{P}$ to improve it. The proofs of the following results can be found in the research report [22].

Remark 3. *For any instance of* Split-Knapsack*, there exists an optimal solution with at most one split item (i.e., at most one $i \in J$ for which $0 < x_i < 1$). Furthermore, this split item, if there is one, has the smallest $\frac{p_i}{s_i}$ ratio.*

Theorem 1. *The decision version of* Split-Knapsack *is NP-complete.*

Theorem 2. *An optimal solution to* Split-Knapsack *can be found in time $O(nsCM)$ with a dynamic programming algorithm.*

Proof. We first assume that the items are sorted by non-increasing $\frac{p_i}{s_i}$ ratios. For any value $0 \leq u \leq C$, $0 \leq l \leq M$ and $0 \leq i \leq n$, let us define $P(u, l, i)$ to be the maximum profit that can be reached with a capacity u, with a memory size at most l, and by using only items numbered from 1 to i, without splitting. We can easily derive that $P(u, l, i+1)$ can be computed as

$$\begin{cases} 0 & \text{if } l = 0 \text{ or } i = 0 \\ \max(P(u,l,i), & \\ \quad P(u - s_{i+1}, l - m_{i+1}, i) + p_{i+1}) & \text{if } u \geq s_{i+1} \text{ and } l \geq m_{i+1} \\ P(u,l,i) & \text{otherwise} \end{cases}$$

We can thus recursively compute $P(u, l, i)$ in $O(nCM)$ time.

Using Remark 3, we can use P to compute $P'(i)$, defined as the maximum profit that can be reached in a solution where i is split:

$$P'(i) = \max_{0 < x < 1} P(C - x s_i, M - m_i, i - 1) + x p_i$$

Computing P' takes $O(nC)$ time. The optimal profit is then the maximum value between $P(C, M, n)$ (in which case no item is split) and $\max_{1 \leq i \leq n} P'(i)$ (in this case item i is split). □

Algorithm 1 Summary of our two-step packing heuristic

function HOMOGENEOUS(B_i)
  Binary Search for X satisfying eq. (9)
  Compute $n_i = g_i(X)$, then A_i according to eq. (7)
  return n_i, A_i
end function
function HEURISTIC
  Compute B_i using z_{r_i} from normal law
  repeat
    $n_i, A_i \leftarrow$ HOMOGENEOUS(B_i)
    Compute n'_i from binomial distribution
    Compute B_i from eq. (7) with n'_i and A_i
  until no B_i has changed by more than ε
  $\mathcal{C} \leftarrow$ greedy configurations
  repeat
    Solve Eq(10) with configurations from $\mathcal{C}$
    Get dual variables p_i
    $c \leftarrow$ solution of SPLIT-KNAPSACK(p_i)
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$
  until Solution of SPLIT-KNAPSACK has profit ≤ 1
end function

In this section, we have proposed a two-step algorithm to solve the allocation problem under reliability constraints. The complete algorithm is summarized in Algorithm 1. The execution time of the first loop is linear in ns, and in practice it is executed at most 10 times, so the first step is linear (and in practice very fast). The execution time of the second loop is also polynomial: solving a linear program on rational numbers is very efficient, and the dynamic program has complexity $O(nsMC)$. Furthermore, in practice the number of generated configurations is very low, of the order of ns, whereas the total number of possible configurations is $O(ns^M)$. It is also interesting to note that although this implementation based on column generation has no polynomial complexity guarantee, it is possible to use the same dynamic algorithm for SPLIT-KNAPSACK in an ellipsoid-based algorithm [23] to obtain a polynomial algorithm. However, in practice the column generation algorithm is much faster.

4. EXPERIMENTAL RESULTS

In this section, we present experimental results for Algorithm 1, where we aim to analyze its performance both in terms of number of machines used and of time complexity. The experiments are based on the Google cluster usage data [14], and more particularly the two datasets LARGE and HPLR identified in Section 2.

Simulations were conducted using a node based on two quad-core Nehalem Intel Xeon X5550, and the source code of all heuristics and simulations is publicly available on the Web at `http://www.labri.fr/perso/eyraud/index.php/Main/CloudAllocation`.

4.1 Resource Allocation Algorithms

In order to assess the performance of our algorithm, it is possible to compute a lower bound on the number of machines that need to be used to satisfy all the reliability constraints. By using Hoeffding bounds for a single service, it is possible to prove [16] that the total cpu usage A_i allocated to service i needs to satisfy $A_i \geq \frac{K_i}{1-f+\sqrt{\frac{-1}{2}\log r_i}}$. The sum of these lower bounds over all services provides the required lower bound. We note that this lower bound does not take memory constraints into account.

In order to give a point of comparison to describe the contribution of our algorithm, we have designed an additional simple greedy heuristic. This heuristic is based on an exclusivity principle: two different services are not allowed to share the same machine. Each service is thus allocated the whole CPU power of some number of machines. The appropriate number of machines for a given service is the minimum number of machines that have to be dedicated to this service, so that the reliability constraint is met. This can be easily computed using the cumulative distribution function of a binomial distribution and a binary search. We greedily assign the necessary number of machines to each service and obtain an allocation that fulfills all reliability constraints. This heuristic is named no_sharing.

In the algorithms based on column generation, the linear program for eq (10) is solved in rational numbers, because its integer version is too costly to solve optimally. An integer solution is obtained by rounding up all values of a given configuration, so as to ensure that the reliability constraints are still fulfilled. We will denote our heuristic by colgen.

4.2 Simulation settings

The experiments are based on both the LARGE and HPLR datasets, as described in Section 2. Both datasets are divided in a number of time intervals (20 for LARGE, 86 for HPLR), and for each of these time intervals, the selected services are described with the total CPU usage and the average memory usage of all of their tasks running at that time. Since reliability requirements are not specified in the trace, we rely on randomly generated values: the requirement of each service is chosen as 10^{-X}, where X is drawn uniformly between 2 and 8. For each time interval, 10 experiments are performed, with 10 different assignements of reliability requirements.

In the Google trace, most of the machines are homogeneous in terms of CPU and memory capacity; we have thus used the characteristics of these machines (0.5 CPU and memory capacity in the normalized units of the trace) in our experiments. Finally, the failure probability of each machine is set to 0.01.

4.3 Number of required machines

The number of required machines by each heuristic is shown on Figure 3 for the LARGE dataset, and on Figure 4 for the HPLR dataset. We can first see that the experiments are quite stable: the variability incurred by the random choices of reliability requirements is rather low. We can also see that colgen uses consistently less than 5% more machines than the lower bound for the LARGE dataset (4.2% on average), and less than 10% for the HPLR dataset (8.2% on average). On the other hand, no_sharing requires significantly more machines, up to 15% more than the lower bound for the LARGE dataset (13.6% on average), and up to 30% more for the HPLR dataset (25% on average).

The fact that both heuristics perform better for the LARGE dataset can be explained by the fact that for services with lower CPU requirements, the approximations of both heuris-

tics are not so precise, and the lower bound is rather optimistic. This can also be seen on time intervals 15 and 17 on the LARGE dataset, where there is actually one very large service which represents a large part of the CPU usage: the solutions are dominated by this service, and both heuristics obtain better results compared to the lower bound.

4.4 Execution time

Since the number of services in both datasets is of the same order of magnitude (around 100 to 200), to analyze the dependency of the execution time of colgen on the number of services, we have generated arbitrarily large instances in the following way. We have merged all the time intervals of the LARGE dataset, and we have generated instances with N services by picking randomly N services in this merged list. Once again, for each value N we generate 10 different instances.

We show on Figure 5 the execution time and the number of configurations generated by the column generation procedure. We can see that the execution time depends on the square of the number of services, which is explained by the fact that the number of configurations generated depends linearly on the number of services (with a slope of about 3.5). Since the complexity of the dynamic program which generates the next configuration is also linear, this yields indeed a quadratic complexity.

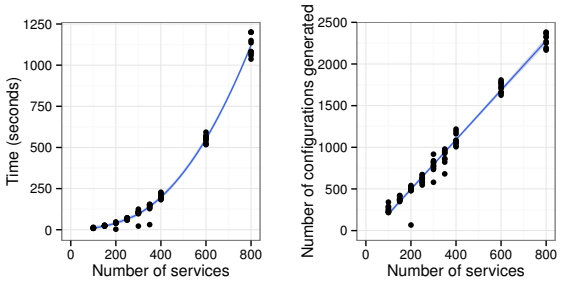

Figure 5: Execution time and number of configurations generated.

5. CONCLUSION

The evolution of large computing platforms makes fault-tolerance issues crucial. In this respect, this paper considers a simple setting, with a set of services handling requests on an homogeneous cloud platform. To deal with fault tolerance issues, we assume that each service comes with a global demand and a reliability constraint. Our contribution follows two directions. First, we analyze a real cluster usage trace to provide a strong justification of the choices and assumptions of our model. Second, we borrow and adapt from the Mathematical Programming and Operations Research literature the use of Column Generation techniques, that enable to solve efficiently some classes of linear programs. These techniques enable us to solve in an efficient manner the resource allocation problem that we consider: the performance of our algorithm is consistently within 10% of a theoretical lower bound. This is done under realistic settings (both in terms of size of the problem and characteristics of the applications, for instance discrete unsplittable memory constraints) and we believe that it can be extended to many other fault-tolerant allocation problems. In particular, it would be interesting to consider a model where the memory usage of a service instance is an affine function of the number of requests it handles.

6. REFERENCES

[1] W. Shih, S. Tseng, and C. Yang, "Performance study of parallel programming on cloud computing environments using mapreduce," in *International Conference on Information Science and Applications (ICISA)*. IEEE, 2010, pp. 1–8.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] F. Cappello, H. Casanova, and Y. Robert, "Checkpointing vs. migration for post-petascale supercomputers," *ICPP'2010*, 2010.

[4] A. Bouteiller, F. Cappello, J. Dongarra, A. Guermouche, T. Hérault, and Y. Robert, "Multi-criteria checkpointing strategies: response-time versus resource utilization," in *Euro-Par 2013 Parallel Processing*. Springer, 2013, pp. 420–431.

[5] C. Wang, Z. Zhang, X. Ma, S. S. Vazhkudai, and F. Mueller, "Improving the availability of supercomputer job input data using temporal replication," *Computer Science-Research and Development*, vol. 23, no. 3-4, pp. 149–157, 2009.

[6] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[7] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the clouds: A berkeley view of cloud computing," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[8] W. Cirne and E. Frachtenberg, "Web-scale job scheduling," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2013, pp. 1–15.

[9] R. Calheiros, R. Buyya, and C. De Rose, "A heuristic for mapping virtual machines and links in emulation testbeds," in *ICPP*. IEEE, 2009, pp. 518–525.

[10] O. Beaumont, L. Eyraud-Dubois, H. Rejeb, and C. Thraves, "Heterogeneous Resource Allocation under Degree Constraints," *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[11] A. Beloglazov and R. Buyya, "Energy efficient allocation of virtual machines in cloud data centers," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 577–578.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[13] D. Hochbaum, *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.

[14] J. Wilkes, "More Google cluster data," Google research blog, Nov. 2011, posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[15] O. Beaumont, L. Eyraud-Dubois, and H. Larchevêque,

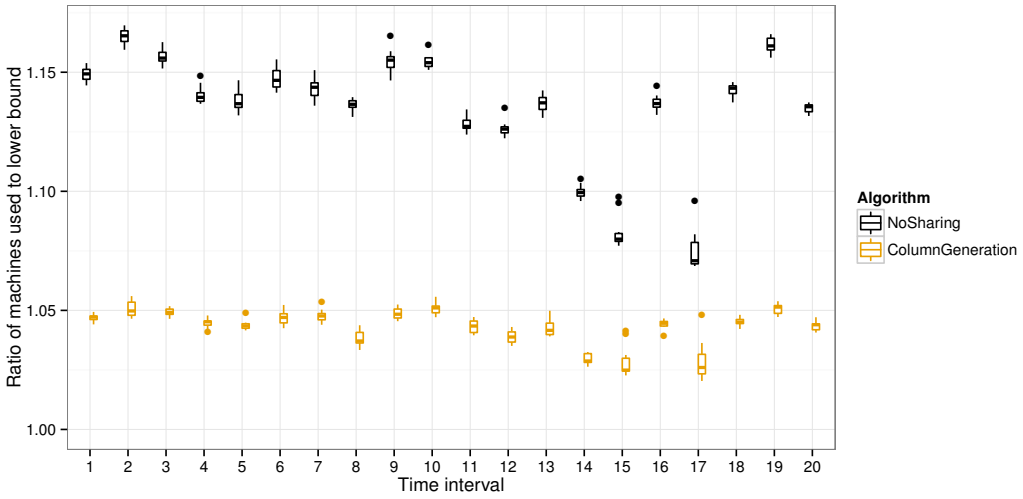

Figure 3: Number of required machines for the Large dataset

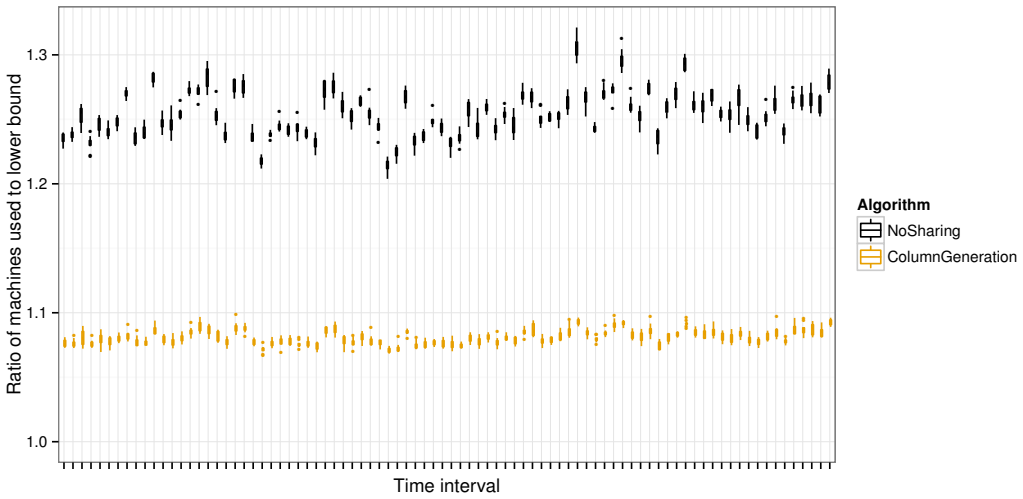

Figure 4: Number of required machines for the HpLr dataset

"Reliable service allocation in clouds," in *IPDPS'13 IEEE International Parallel & Distributed Processing Symposium*, 2013.

[16] O. Beaumont, P. Duchon, and P. Renaud-Goud, "Approximation algorithms for energy minimization in cloud service allocation under reliability constraints," in *HIPC'2013, IEEE international conference on High Performance Computing, Bangalore*, 2013.

[17] L. Valiant, "The complexity of enumeration and reliability problems," *SIAM J. Comput.*, vol. 8, no. 3, pp. 410–421, 1979.

[18] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011, revised 2012.03.20. Posted at URL http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.

[19] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *ACM Symposium on Cloud Computing (SoCC)*, San Jose, CA, USA, Oct. 2012.

[20] O. Beaumont, L. Eyraud-Dubois, P. Pesneau, and P. Renaud-Goud, "Reliable service allocation in clouds with memory and capacity constraints," in *Resilience – EuroPar workshop*, 2013.

[21] J. Desrosiers and M. E. Lübbecke, *A primer in column generation*. Springer, 2005.

[22] O. Beaumont, L. Eyraud-Dubois, J.-A. Lorenzo, and P. Renaud-Goud, "Efficient and robust allocation algorithms in clouds under memory constraints," Inria Bordeaux Sud-Ouest, Tech. Rep., May 2014. [Online]. Available: http://hal.inria.fr/hal-00874936

[23] G. L. Nemhauser and L. A. Wolsey, *Integer and combinatorial optimization*. Wiley New York, 1988, vol. 18.