Optimizing Resource allocation while handling SLA violations in Cloud Computing platforms

Lionel Eyraud-Dubois, Hubert Larchevêque

Optimizing Resource allocation while handling SLA violations in Cloud Computing platforms

Lionel Eyraud-Dubois, Hubert Larchevêque
INRIA Bordeaux – Sud-Ouest
University of Bordeaux
{lionel.eyraud-dubois|hubert.larcheveque}@labri.fr

***Abstract*—In this paper, we study a resource allocation problem in the context of Cloud Computing, in which a set of Virtual Machines (VM) has to be allocated on a set of Physical Machines (PM). Each VM has a given demand (e.g. CPU demand), and each PM has a capacity. However, VMs only use a fraction of their demand. The aim is to exploit the difference between the demand of the VM and its actual resource usage, to achieve a higher utilization on the PMs. However, the resource consumption of the VMs might change over time (while staying under its original demand), implying sometimes expensive "SLA violations" when the demand of some VMs is not satisfied because of overloaded PMs. Thus, while optimizing the global resource utilization of the PMs, it is necessary to ensure that at any moment a VM's need evolves, a few number of migrations (moving a VM from PM to PM) is sufficient to find a new configuration in which all the VMs' consumptions are satisfied. We model this problem using a fully dynamic bin packing approach and we present an algorithm ensuring a global utilization of the resources of 66%. Moreover, each time a PM is overloaded, at most one migration is sufficient to fall back in a configuration with no overloaded PM, and at most 3 different PMs are concerned by required migrations that may occur to keep the global resource utilization correct. This allows the platform to be highly resilient to a great number of changes.**

I. Introduction

A Cloud Computing platform consists in a set of Physical Machines (PM) onto each of which several Virtual Machines (VM) may be run. Each Virtual Machine comes with a resource demand, for which a guarantee of resource availability has to be ensured by the provider at any moment if the VM really needs it (see [1] for an introduction of the trends of Cloud Computing). On such platforms, the provider has to find the best way to allocate VMs onto PMs in order to exploit as much as possible the resources offered by the platform, while still maintaining the resources availability guarantees. In practice, the real use of the reserved resources is quite low, and those unused reserved resources can be used to run other Virtual Machines, greatly improving the global resource utilization of the complete system. This is the idea of server consolidation, which is studied in this paper.

The main risk of using reserved resources to run other Virtual Machines is that the reserved demand has to be served to each VM when it really requires it, since the consumer has paid for it. Since the VM load can vary quickly and, as considered in this paper, unpredictably, it may happen that the demands of some VMs suddenly becomes unsatisfied. When a VM's demand is not satisfied, it is said to suffer a SLA violation. SLA violations are the downside of server consolidation, and often represent a financial cost for the provider : we could think of a price to pay to each VM concerned by a SLA violation, or for each time slot a VM spend in SLA violations during a given amount of time, ...

In this context, the aim is to develop algorithms that provide a good balance between resource utilization and SLA violations. Some works have explored the use of techniques to predict the VMs' resource consumptions to be more efficient [2], [3]. For example, in [4] the authors try to identify sets of VMs whose consumptions peaks happen at different times in order to allocate them together on the same PMs. This kind of method is out of the scope of this paper, and can be seen as an additional brick that could be plugged onto the kind of algorithms we propose, in order to make them even more efficient.

In Cloud computing systems, live migration techniques allow to modify the allocation of the VMs during their execution, by effectively moving them from one PM to another PM. There exists different strategies to perform VM's migrations, like pre-copy or post-copy [5], [6], but studying their different behavior and modeling their cost is out of the scope of this paper. In this paper, we are interested in providing resource allocation algorithms which limit the number of migrations necessary to handle the platform.

In this context, it is common to consider that four parameters have to be optimized when dealing with resource allocation, while being a priori not compatible :

- **Reactivity :** Optimizing reactivity corresponds to minimizing the time spent in SLA violation after a change in the load of a PM. This might require monitoring each change in the load of the VMs, in order to react as soon as a violation occurs, and, then, to move as quickly as possible enough VMs on the right PMs to return to a valid situation. The main drawback of a reactive approach is the "ping-pong" like modifications in the platform : if the load of a VM load suddenly increases, a highly reactive system will probably migrate it to another PM. Suppose now that, just after this migration,

the load of the target PM increases as well, the same VM might have to be moved again. A less reactive system would certainly have avoided some of those migrations.

- **Robustness :** Optimizing robustness is a bit less precise, but informally it corresponds to producing allocations on which modifications on the load of the VMs have as little impact as possible, so as to minimize the number of SLA violations encountered without doing any migration. Robustness is in some way in opposition with resource optimization, since it usually involves reserving some extra PM capacity to handle load bursts. As an example, the most robust allocation possible is one which reserves for each VM its original demand, but this comes at a high price in terms of resource utilization.
- **Resource Optimization :** Optimizing resources corresponds to minimizing at each PM the resources not effectively used by a VM. This parameter is usually quantified by the ratio between the sums of the load of the VMs on a platform, and the sum of the resources offered by the PMs on this platform. This ratio is always lower or equal than 1.
- **SLA violation :** SLA violations are a major concern in the context of Cloud Computing on a server-side. The problem of handling SLA violations can be considered as the problem of minimizing the number of SLA violations happening at a VM/application/platform during a certain amount of time, or the total time spent in SLA violation by a VM/application/platform during such a period. It can also be considered as the problem of minimizing the time needed by a pair platform/algorithm to handle a SLA violation, the time needed to reach back a stable state.

Note that the main aim is to maximize the resource allocation ratio while optimizing the SLA violation parameter, whatever the policy used to define it. Robustness and reactivity are two parameters that design two opposite types of approaches, with a whole set of possible approaches mixing those two parameters in different proportions. Intuitively, augmenting robustness may imply lower resource optimization, if robustness is ensured by keeping at each PM an amount of resource to handle VM's load variations, but might be useful to minimize the number of SLA violations to handle. On the contrary, improving reactivity allows to handle quickly the SLA violations, what could allow a better use of the resources offered by a platform.

This paper is organized as follows : in Section II we present some related works. In Section III, we present some definitions and the modeling we use for the study of the problem. Section IV is the core part of the paper, in which we present the algorithm and the proofs of its different properties. Section V presents some practical details and perspectives, and Section VI concludes the paper.

II. Related Works

Our work in this paper follows a long line of works on the classical Bin Packing problem [7]. More precisely,we are interested in an online variant of bin packing [8]. In the classical version of online bin packing, items are to be packed one by one as they arrive, and no information is known about the future incoming items. In such a setting, the solutions proposed are compared to an "offline" optimal solution, that can be considered as knowing everything about what is going to happen.

To measure the quality of a solution produced by an algorithm $\mathcal{A}$, we use the classical definition of an approximation ratio $R(\mathcal{A})$ defined as

$$R(\mathcal{A}) = \lim_{n\to\infty} \sup_{OPT(L)=n} \frac{\mathcal{A}(L)}{OPT(L)}$$

where $\mathcal{A}(L)$ and $OPT(L)$ denote, respectively, the number of bins used for packing the list L using algorithm $\mathcal{A}$ or using an offline optimal solution.

Several works have considered dynamic bin packing [9], [10], [11], a setting in which items may arrive and depart at arbitrary times, but they cannot be moved to another bin once they have been assigned, what strongly restricts the quality of the packing. In [10], it is showed that no online algorithm for dynamic bin packing can have a better ratio than 2.428, and in [11] that the ratio of the classical first-fit online algorithm for dynamic bin packing is greater than 2.5. In the context of Cloud Computing, where the technical possibility of live migration exists, we can design a model which allows to move items from bin to bin, and having this capacity helps us to outperform the solutions they proposed, since we present in this paper an algorithm ensuring an approximation ratio of $\frac{3}{2}$.

In fact, the problem studied in this paper is very close to the fully dynamic bin packing problem [12], which is a variant of bin packing where :

- items can arrive and depart from the packing dynamically,
- items may be moved from bin to bin as the packing is adjusted to accommodate arriving and departing items.

In this paper, we also consider the slightly different context in which the item sizes may change dynamically. Note that a variation of an item's size cannot be simulated by removing this item and adding it with its new size. Indeed, when the item is added again with a larger size, the packing algorithm has to repack this item in a different bin. However when an item's size changes, a packing algorithm can choose to move some other items from the same bin, and that cannot be taken into account when considering only arrivals and departures of items. Similarly, when the size of an item decreases, it could be possible to keep the packing as it is, whereas removing and reinserting it might involve much

more changes. On the other hand, it is possible to simulate arrivals and departures of items by considering that their sizes evolve from or to a size of 0.

In order to measure the performance of an algorithm for our bin packing variant, we consider the number of migrations used by this algorithm to fully process a change (an insert, delete or a change in an item's weight). In [12] the authors presented a $\frac{5}{4}$ approximation algorithm for fully dynamic bin packing (*i.e.* only considering arrival and departures of items), with a $O(1)$ bound on the maximal number of moves required, but the constant itself is very large. In [13] authors are interested in the same variant, and present a 1.33 approximation algorithm in which the number of moves is upper bounded by 7. Moreover, these results are obtained with the assumption that an arbitrary large number of very small items can be considered as one "group" of items if its overall size remains small, and that these groups can be moved as if they were one single item, only counting for one move. In the same context, the number of moves required by the algorithm we present is at most 6.

III. Model

As explained in the previous Section, we model the problem using a classical bin packing approach, in which each VM is modeled as an item having a size (or weight) in $(0, 1]$, and each PM is modeled as a bin with an overall capacity of 1. The bin packing problem can be defined as follows :

Definition 3.1: Given a set of items $I = \{i_1, \ldots, i_m\}$, with weights $w(i) \in [0, 1) \forall i \in I$, find a collection of disjoint subsets $b_1, \ldots b_n$ of I called bins, of minimal cardinality n such that $\forall j \leq n, \sum_{i \in b_j} w(i) \leq 1$.

In this paper, we take into account only one kind of resource demand, *i.e.* only CPU demand for example. This can be realistic in a context where CPU is the only limiting factor for resource allocation. In most of the cases though, other resources like memory or I/O consumption need to be taken into account. However, the goal of this paper is to propose a way to address the problem with a more theoretical approach, that has to be adapted to each practical case encountered. We discuss extensions of our work to the multi-dimensional case in Section V.

Note also that we do not place an a priori bound on the number of PMs available to host the VMs. This assumption is common in the context of bin packing. In practice, when dealing with a fixed number of PMs, the platform provider needs to implement a policy to decide which VMs to accept into the system, which can be a compromise between the optimization about resource utilization and the SLA violation which may arise fro ma high load. However, the design of such policies is out of the scope of this paper and is left for future works.

The context of this study is to consider that an assignment of items to bins already exists, and an event occurs: either the weight of an item evolves (increase or decrease), or a new item has to be placed, or an item disappears.

Definition 3.2 (Correct configuration): A correct packing configuration is an assignment of items to bins in which each bin holds a weight lower than 1.

Since items' weight evolve, the overall weight of the bins also vary. An event occurs either when the overall weight of a bin is goes beyond 1 (overload event) or when its overall weight is considered too small (underload event). At each event, some items can be moved from bin to bin. Such a move is called a migration.

Since many migrations can occur, even for handling one same event, we will say that two migrations are *independent* from each other if they can be performed in parallel with no risk of entering an incorrect configuration. Two migrations are not independent if, for example, the destination bin b of the first one is the same as the source bin of the second one. In such a case, it might happen that performing the first migration before the second one overloads b and makes the configuration incorrect between the two migrations.

IV. A $\frac{3}{2}$-asymptotic approximation algorithm for handling server consolidation

In this section, we present a $\frac{3}{2}$-asymptotic approximation algorithm for handling server consolidation on a homogeneous Cloud Computing platform. As stated in Section III, we consider that VMs have only one kind of resource demand/utilization, and, thus, only one type of characteristic for each PM.

In the context of this paper, there are two main reasons for minimizing the number of violations: the first one is to limit resource usage – migrations are very stressful to the network, so it is important to perform as few migrations as possible to allow the system to handle a high variability of the VM load. The second reason is that migrations take time, and as long as the necessary modifications are not finished, some VMs might keep experiencing an SLA violation. Both of these reasons lead to slightly different measures of the number of migration – we can count the total number of migrations for a given event, and we can count the number of migrations before getting into a correct configuration. As will be seen in the remainder of the section, the algorithm we propose optimizes both measures.

In the context described in Section III, we prove the following theorem :

Theorem 4.1: Algorithm 1 ensures that each time an overload event occurs, it performs one migration to obtain a correct bin packing configuration. Moreover, it performs additional migrations to reach a new configuration into which an asymptotic approximation ratio of $\frac{3}{2}$ is ensured. All migrations linked to a given event in order to build a new compact configuration concern at most 3 different bins, and are independent from each other.

Proof: Theorem 4.1 is a combination of Lemma 4.2, Lemma 4.3, Lemma 4.5 and Lemma 4.4 that are proved later in this paper. ■

A. Algorithm

In this section, we describe Algorithm 1. To describe it, we divide items into different types based on their sizes :

- B-item : item i for which $w(i) \in (\frac{2}{3}, 1]$
- L-item : item i for which $w(i) \in (\frac{1}{2}, \frac{2}{3}]$
- S-item : item i for which $w(i) \in (\frac{1}{3}, \frac{1}{2}]$
- T-item : item i for which $w(i) \in (0, \frac{1}{3}]$

We will denote each bin by a notation corresponding to the set of items it contains. For example a B-bin contains one B-item, while an SS-bin contains two S-items, and a $LT*$-bin contains one L-item and possibly several T-items. A particular case is the case of $T*$-bins. A $T*$-bin will be named as "filled" if its overall weight is greater or equal to $\frac{2}{3}$, and "unfilled" otherwise. Those unfilled $T*$-bins will be denoted by UT.

The algorithm we propose performs additional migrations to obtain a compact configuration, defined as follows:

Definition 4.1 (Compact configuration): A *compact packing* configuration is a correct configuration in which there exists at most one unfilled $T*$-bin, at most one S-bin, and in which there is not at the same time an unfilled $LT*$-bin and an $ST*$-bin or a $T*$-bin.

Algorithm 1 is applied each time an event occurs, either an overload event, *i.e.* the overall weight of a bin is growing above 1, or an underload event, which happens when the configuration is not compact anymore. In such an approach, many changes in the items' weights may occur before an item move is really needed, since changes of an item's weight does not necessarily imply an overload or underload event.

In case of an overload event, Algorithm 1 is working in two phases: in a first phase, at most one item is moved from the overloaded bin in order to obtain a correct configuration. In a second phase, additional items are moved in order to obtain a compact configuration. The structure of the bins implied by the compactness of the resulting configuration, as will be shown, allows us to ensure an asymptotic approximation ratio of $\frac{3}{2}$.

First, we present the different primitives needed by Algorithm 1. x denotes any item on which the procedure can be applied. cb denotes the current bin (to which x is allocated), whereas b denotes another bin, s denotes a small item (the procedure insert() only applies to small items) :

- `new(x)` or `new(x,y)` : Open a new bin and put x (respectively x and y) in it. This corresponds to switching on a new PM, or use a PM that was not already in use.
- `move(x,b)` : Move item x to bin b. This procedure corresponds to a migration of x from cb to b.
- `insert(s)`

```
1: if ∃b a S-bin then
2:     move(s, b)
3: else
4:     new(s)
5: end if
```

- `fill(b)` : This procedure is used to fill a $LT*$-bin with T-items so that it reaches a weight greater than $\frac{2}{3}$ when possible.

```
1: while w(b) ≤ 2/3 and (∃t a T*-bin or ∃st a ST*-bin) do
2:     if ∃ut a UT-bin then
3:         move(x, b), x ∈ ut
4:     else
5:         if ∃t a T*-bin then
6:             move(x, b), x ∈ T*-bin.
7:         else
8:             move(x, b), x ∈ ST*-bin.
9:         end if
10:    end if
11: end while
```

- `merge(`ut_1`,`ut_2`)` : This procedure is used to merge two UT-bins into one, ensuring that at most one of them remains unfilled (the other is either empty or filled).

```
1: ut ← ut_i ∈ (ut_1, ut_2) such that
   w(ut_i) = min(w(ut_1), w(ut_2))
2: t ← ut_j ∈ (ut_1, ut_2) such that i ≠ j
3: while w(ut) > 0 and w(t) ≤ 2/3 do
4:     move(x, t), x ∈ ut
5: end while
```

In the following, we present Algorithm 1, which is run when an event occurs at bin cb. An overload event occurs when $w(cb) > 1$. In such a situation, the greatest item of the current bin is moved to a new bin (or to an S-bin or a UT-bin if it is a S-item or a T-item, respectively).

An underload event is said to occur if $w(cb) < \frac{2}{3}$. If cb is composed only of T-items and its weight is under $\frac{2}{3}$, it is a UT bin that can be kept in this state if no UT bin already exists. In a underload situation, if the main item is a L-item, the $fill()$ procedure is used to fill the current bin, if it is a S-item, the procedure $insert()$ is used to move it, and if there are only T-items in the bin, it has to be merged with the possibly already existing UT-bin.

Note that if cb is overloaded at first, it can become underloaded after some of the migrations performed by Algorithm 1, but the contrary is not true (assuming that the weights of the items do not evolve during the execution of Algorithm 1). Thus Algorithm 1 handles underload events

after having handled overload situations. Note that it is possible to postpone the actual execution of the migrations until all the required moves for a given event have been computed.

while $w(cb) > 1$ **do**
  if $\exists b \in cb$ a $B - item$ **then**
    $new(b)$
  else if $\exists (s_1, s_2) \in cb$ two $S - items$ **then**
    $new(s_1, s_2)$
  else if $\exists l \in cb$ a $L - item$ **then**
    if $\exists s \in cb$ a $S - item$ and $w(cb) - w(s) \leq 1$ **then**
      $insert(s)$
    else
      $fill(new(l))$
    end if
  else if $\exists s \in cb$ a $S - item$ **then**
    $insert(s)$
  else
    find x with $w(x) = \max_{e \in cb}(w(e))$
    if $\exists ut$ a $UT - bin$ **then**
      $move(x, ut)$
    else
      $new(x)$
    end if
  end if
end while
while $w(cb) \leq \frac{2}{3}$ **do**
  if $\exists l \in cb$ a $L - item$ **then**
    $fill(cb)$
  else if $\exists s \in cb$ a $S - item$ **then**
    $insert(s)$
  else if $\exists ut$ a $UT - bin$ and $cb \neq ut$ **then**
    $merge(cb, ut)$
  end if
end while

Algorithm 1: Handling overload and underload events occuring at bin cb

B. Proofs of the asymptotic approximation ratio

We first prove that at the end of the execution of Algorithm 1, the configuration is always compact. Then we prove that in a compact configuration, the asymptotic ratio is $\frac{3}{2}$.

Lemma 4.2: After an event has been detected in a configuration that was compact, and after Algorithm 1 has been executed to solve it, the bins are in a compact configuration if no new event has occurred.

Proof: First of all, since no new event occurs, at the end of the execution of Algorithm 1, there exists at most one S-bin, otherwise an "$insert()$" action would have been performed, that would have merged the two S-items together. There also exists at most one UT-bin, otherwise a "$merge()$" action would have been performed.

Now suppose that there exists an unfilled $LT*$-bin. By definition of the procedure "$fill$", there is no $ST*$-bin and no $T*$-bin in the configuration. Hence the configuration is compact. ■

Lemma 4.3: In a compact configuration, at most $\frac{3}{2}OPT(\mathcal{L}) + 2$ bins are used, where $OPT(\mathcal{L})$ denotes the number of bins used for packing the list $\mathcal{L}$ in an optimal solution.

Proof: Consider a compact configuration $\mathcal{C}$, like the one obtained using Algorithm 1. We divide the proof in two cases, depending on the presence of $T*$-bins in $\mathcal{C}$:

- If there exists a $T*$-bin (filled or unfilled) in $\mathcal{C}$, then there exists at most one UT-bin, and at most one S-bin whose overall weight is strictly less than $\frac{2}{3}$ (hence the additional 2 term in the lemma statement). Moreover, since there does not exists any unfilled $LT*$-bin, all bins except maybe 2 have weight at least $\frac{2}{3}$. This yields the announced approximation ratio when the number of bins is sufficiently large.
- If there exists no $T*$-bin in $\mathcal{C}$, a first subcase is when all bins have weight at least $\frac{2}{3}$ except for at most one S-bin. Then the approximation ratio is proved.
  If some bins have weight lower than $\frac{2}{3}$, such bins are necessarily $LT*$-bins that have not been filled enough to reach $\frac{2}{3}$. Indeed, all other types of bins have weight at least $\frac{2}{3}$. Note that there cannot be any $ST*$-bin by definition of a compact configuration.
  Since such $LT*$-bins are unfilled, and since there are no $T*$-bin and no $ST*$-bin, we can remove all existing T from the list $\mathcal{L}$ without changing the number of existing bins in $\mathcal{C}$. We denote this new list $\mathcal{L}'$, and it is easy to see that $OPT(\mathcal{L}') \leq OPT(\mathcal{L})$. As before, there is at most one S-bin, which we ignore for now and will count at the end.
  To perform the analysis, we need to consider the possible structure of the bins present in $\mathcal{C}$, *i.e.* the set $\{B, L, LS, SS\}$ (since we removed the T-items), which is also the set of possible structure of the bins of an optimal solution on the same set of items.
  Now we denote by $n_{\mathcal{C}}(B)$ the number of B-bins in $\mathcal{C}$, and by $n_{OPT}(B)$ the number of B-bins built by an optimal solution. We extend this notation to all types of bins.
  Note that an optimal solution does not use several S-bins, otherwise they could be merged together. If it uses one, we can also disregard it (and count it in the additional term at the end). Since both solutions contain all of the items, we have the following equations :

$$n_{\mathcal{C}}(B) = n_{OPT}(B)$$
$$n_{\mathcal{C}}(L) + n_{\mathcal{C}}(LS) = n_{OPT}(L) + n_{OPT}(LS)$$
$$n_{\mathcal{C}}(LS) + 2n_{\mathcal{C}}(SS) = n_{OPT}(LS) + 2n_{OPT}(SS)$$

  Summing all these equations to compute the number of

bins used in $\mathcal{C}$, we obtain :

$$\begin{aligned} n_{\mathcal{C}} &= n_{\mathcal{C}}(B) + n_{\mathcal{C}}(L) + n_{\mathcal{C}}(LS) + n_{\mathcal{C}}(SS) \\ &\leq n_{OPT}(B) + n_{OPT}(L) + \frac{3}{2} n_{OPT}(LS) + n_{OPT}(SS) \\ &\leq \frac{3}{2} n_{OPT} \end{aligned}$$

This yields the lemma statement, and the asymptotic approximation ratio of $\frac{3}{2}$.

■

C. Impact on migrations

In this Section, we analyze the migration cost of Algorithm 1, in three different ways. We first bound the number of migrations required to reach a correct configuration in Lemma 4.4. Then we prove in Lemma 4.5 that at most 3 bins are impacted by the migrations associated to a given event. Finally, we provide in Lemma 4.6 a bound on the total number of migrations performed when considering that small items can be moved together (which is a common assumption in dynamic bin packing).

Lemma 4.4: At any moment, when an event has to be handled, Algorithm 1 needs to perform at most one migration to obtain a new correct configuration.

Proof: Note that the algorithm reacts after the change of size of one item. This means that when the weight of a bin gets above 1, there exists one item in the bin whose weight has just increased.

After the weight modification in the bin, if the bin contains a B, a L or a S-item, it is sufficient to move it in this order of preference. The exception is for the case when the bin ends in the $LST*$ configuration, where it might be more efficient to move the S instead of the L-item, if moving the S-item is sufficient to reduce the bin's weight under 1. Even in that case, moving one item is sufficient. If the problematic bin only contains T-items, moving the largest one is enough to get the bin weight under 1, since either it is the growing one, or it has a larger weight than it. ■

Lemma 4.5: At any moment, when an event has to be handled, all the migrations Algorithm 1 needs to perform are independent. Moreover, all the migrations concerning the same event concern at most 3 different bins (plus the considered one).

Proof: All the migrations Algorithm 1 need to perform concern either a new bin, a S-bin or a UT-bin. We first analyze these three cases in more detail.

Moving an item from the current bin to a new bin does not impact any other migration, and concerns one bin (the new one). No more than one new bin is necessary to handle an event, into which the largest items are placed.

Moving an item from the current bin to a S-bin does not impact any other migration either, and also concerns at most one bin (the S-bin). At most one S-bin is concerned for an event, since if two S-items are to be moved, they are moved together in a new bin. It is possible that three S-items have to be moved if moving the two first ones in a new bin leave the current bin in an underloaded situation. In such a case, the last S-item has to be moved in the S-bin if it exists. But still, this last migration is the only one that may concern the S-bin. No more than three S-items can be in the same bin when an event occurs, otherwise an event would have occurred in a previous step (since a bin which contains 4 items, three of them being S-items, has necessarily a weight larger than 1).

A careful analysis of Algorithm 1 shows that at most two migrations of the type "move to a new bin" or "move to a S-bin" are necessary to handle a given event. Indeed, a first one might be necessary to handle the overload event, but in that case no other such migration is necessary. A second such migration may be required to handle the underload situation if one occurs in the current bin.

When filling a $LT*$-bin, the only other bin concerned is the UT bin. However this UT-bin might not contain enough T-items to fill the current bin to a weight greater than $\frac{2}{3}$. In such a case, an existing filled $T*$-bin, if such a bin exists, has to be used as if it was a UT-bin, even if its current weight is still greater than $\frac{2}{3}$. T-items will be taken from this bin to fill the current $LT*$-bin.

In some cases, like when handling an overloaded $LLT*$-bin, a new bin containing one of the L-items needs to be filled, and the current bin may also need to be filled with some additional T-items. In the worst case, an overall weight of at most $\frac{1}{3}$ has to be moved ($\frac{1}{6}$ for each "fill" action). In such a case, even if the first UT-bin considered does not contain enough T-items, since the second T-bin is necessarily filled, and thus has weight at least $\frac{2}{3}$, then it is sufficient to consider those two T-bins to fill both $LT*$-bins.

The only other action in which T-items are moved is when merging two UT-bins, the current bin being one of them. Such a merge concerns only the already existing UT-bin.

We now count the maximal number of different bins concerned by the resolution of an event.

When handling an overload event, if a T-item is moved, the current bin cannot become underloaded. Thus, at most the target UT-bin is concerned for the overall resolution.

In all other cases, at most one bin is concerned by the resolution of the overload event (either a new one, or the S-bin). If the current bin does not become an underloaded $ST*$-bin, a "$fill()$" or a "$merge()$" action may be required, concerning at most two more bins for a total of 3 concerned bins (remember that even if handling the overload event requires to fill the newly created bin, two "$fill()''$ actions only concern at most two $T*$-bins).

If the current bin becomes an underloaded $ST*$-bin after the first migration, another new bin (or the current S-bin) might be concerned by the subsequent "$insert()$" action. In such a case, the first migration did not concern a S-

item (otherwise both would have been moved together). If it concerned a B-item, then the only work still to be done is to merge what is left in the current bin with the UT-bin. The overall number of concerned bins is 3.

The last case (the first migration concerns a L-item) is not possible. Indeed, by definition of Algorithm 1, this would mean that $w(L) + w(T*) \geq 1$. Since $w(L) \geq \frac{1}{2}$, this would mean that $w(T*) \geq \frac{1}{2}$, and thus, because $w(S) \geq \frac{1}{3}$, that $w(S) + w(T*) \geq \frac{5}{6}$, in which case the current bin would not be in an underloaded situation.

Note that the computation of the number and location of migrations can be performed before beginning to handle them Thus all migrations are independent, and at most 3 bins are implied. ■

Note on the total number of migrations Since the size of T-items can be arbitrarily small (as long as they remain positive), the number of T-items in a bin is not bounded. In consequence, it is not theoretically possible to provide a bound for the total number of migrations required by Algorithm 1 to obtain a compact configuration. Indeed, the "$merge()$" action may require an unbounded number of moves.

However, it is common in the literature to consider [13] that T-items can be grouped in the following way :

Definition 4.2 (T-items grouping): In any bin, T-items are divided into non-overlapping groups such that all groups have size less than $\frac{1}{3}$, but the sum of the weights of any two groups from the same bin is larger than $\frac{1}{3}$.

Using this grouping technique, the number of migrations induced by Algorithm 1 can be upper bounded, considering that one T-items group accounts for only one migration.

Lemma 4.6: Algorithm 1 moves at most 6 groups/items for each event it solves.

Proof: We start by considering the number of groups' migrations involved when merging two UT-bins. Note that this consists in identifying the most loaded one, and moving the items of the other one into it. If only one of the two bins has weight more than $\frac{1}{3}$, merging the other one into it requires at most one migration. If both have weight more than $\frac{1}{3}$, at most 3 groups need to be moved from one bin to the other, since moving 4 groups from a bin would involve an overall weight greater than $\frac{2}{3}$, and would overload the destination bin. Thus the fusion of two UT-bins involves at most 3 migrations.

We now consider the number of migrations involved when filling a L-bin (or a $LT*$-bin). Since the weight of an L-item is at least $\frac{1}{2}$, by the group definition moving two groups from the UT-bin is enough to perform the "$fill()$" operation. However, the considered UT-bin may not contain enough T-items, and may contain only one group. In that case, after having moved the only group in this bin, another T-bin is chosen, and two groups from this bin are enough to fill the $LT*$-bin. Overall, at most 3 groups are moved.

Handling an underloaded $ST*$-bin involves at most one migration to move the S-item away, followed by the fusion of two UT-bins, thus involving at most 4 moves. Handling an underloaded $T*$-bin requires only 3 moves, and the case of an underloaded $LT*$-bin requires at most 3 moves to fill the bin.

Using the same reasoning as in proof of Lemma 4.5, we observe that the worst case, *i.e.* the configuration of an overloaded bin in which the maximum number of migrations is required, is when Algorithm 1 has to deal with an overloaded $LLT*$-bin. In this case, one migration is necessary to place an L-item on an empty bin. The worst case happens when the current bin becomes unfilled, which requires to fill two bins. In such a case, as proved for Lemma 4.5, only two T-bins have to be considered, the current UT-bin and possibly another one if the UT-bin does not contain enough items. Filling each bin with weight at least $\frac{1}{3}$ is achieved as soon as two groups from the same $T*$-bin are moved. Thus at most 5 groups from two different T-bins are moved to fill both LT-bins. Adding the first migration of the L-item yields that, when using this grouping technique, at most 6 moves are necessary for Algorithm 1 to handle any event. ■

As stated in the introduction, this grouping trick is relevant when items represent physical items which can be moved, since in that case it is often indeed possible to pick several small items at the same time. In the context of Cloud computing, though, the cost associated to the migration of a VM is not *a priori* correlated with its CPU consumption, so there is no real reason to consider that small items are somewhat easier to move.

Another, more appropriate approach to count the number of moves is to assume that the CPU consumption of an active VM can be lower bounded. In practice, VMs with arbitrarily small CPU consumption do not exist: it is possible to separate the VMs using a value $w_{\min} \in (0, \frac{1}{3})$ under which a VM can be considered as *sleeping*, and above which a VM is active (we consider this value lower than $\frac{1}{3}$ otherwise there is no T-item to consider !). It is reasonable to consider that a sleeping VM does not need to have its CPU consumption served with a critical SLA, since it just needs to be "kept alive". Thus it makes sense to assume that its CPU consumption can be disregarded.

If we consider that each T-item has a weight of at least $w_{\min}$, then there can be at most $c_{\min} = \lfloor \frac{1}{w_{\min}} \rfloor$ T-items in a given bin.

The type of underload event in which Algorithm 1 has to move the greatest weight of T-items is when handling an underloaded $T*$-bin, which requires to move an overall weight of at most $\frac{2}{3}$. Moving a set of T-items with an overall weight of $\frac{2}{3}$ requires at most $\lfloor \frac{2}{3} c_{\min} \rfloor$ migrations.

When considering overload events, a fine analysis of Algorithm 1 (that is omitted here due to lack of space) shows that in each configuration, the weight of remaining T-items can be upper bounded. For example, when dealing

with a $BBT*$-bin, the item whose weight increased last is a B-item, since no two B-items can stand in the same bin. Thus, since a B-item and the same set of T-items did not overload the bin before this load increase, the weight of the set of T-items can be upper bounded by $\frac{1}{3}$ (since the weight of a B-item is at least $\frac{2}{3}$).

The same reasoning can be used for each configuration, and, using the reasoning of the proof of Lemma 4.5, we conclude that the configuration which requires the largest number of migrations is in the case of an overloaded bin of the type $LLT*$. In this case, two L-items need to be moved from the current bin. By the reasoning described before, It is possible to show that the overall weight of the T-items is at most $\frac{1}{2}$. Two "$fill(L)$" actions have to be performed, consisting in filling the new L-bins built upon the L-items. Each $fill(L)$ action requires to move an overall weight of at most $\frac{1}{6}$. Hence, the two $fill(L)$ actions require the migration of at most $2\lfloor\frac{1}{6}c_{\min}\rfloor$ items. In conclusion, the maximum number of migrations required to handle an overload event (and any event, in fact) is $1+2\lfloor\frac{1}{6}c_{\min}\rfloor$ when the weight of each item is lower bounded by $w_{\min}$.

V. Practical considerations and perspectives

A. Handling simultaneous events

Throughout the paper, we analyze Algorithm 1 by considering that no item size changes while a given event is being solved. In practice, depending on how quickly the item sizes evolve, a second event might occur before the migrations linked to the first one have been completely performed. However, even in that case, the migration performed by Algorithm 1 to reach a correct configuration can be performed immediately, without having to wait for migrations from the previous event to finish.

Moreover, the structure of Algorithm 1, in which decisions are depends only on the content of the current bin, and its parallel property as proven in Lemma 4.5 allows to use it easily in a distributed setting. Indeed, it only requires a distributed structure which allows each PM to quickly identify a UT-PM, a S-PM and an empty PM. Moreover, it is possible to slightly generalize Algorithm 1 to allow more than one UT-PMs and S-PMs at the same time. Indeed, even allowing a small number (logarithmic, for example) of these under-filled PMs would make such a distributed structure much easier and much more efficient, while keeping the approximation ratio on resource utilization relatively small. Having a distributed resource management system is desirable for many reasons (scalability and fault tolerance, for example). In addition to that, having more than one UT-bin would also makes Algorithm 1 even more resilient to a high frequency of events. Indeed, two UT-bins and two S-bins make it possible to handle simultaneously and independently two different events. It would actually be possible to increase the number of unfilled bins when the load variability gets too high, and to decrease it afterwards to improve the resource utilization.

B. Extension to multiple dimensions

In practice, there is often more than one critical resource in the system. In addition to CPU consumption, the allocation of VMs might also be constrained by the amount of available memory on the PMs, and by the amount of storage or I/O capacity. Modeling these additional constraints yields multi-dimensional formulations, in which we define a configuration to be correct if the total demand on a machine does not exceed its capacity, for every dimension. Instead of bin packing, this requires to solve the much more difficult vector packing problem [14], which has been proved to not accept any APTAS as soon as two dimensions are considered. Chekuri *et al.* presented in [15] a $1+d\epsilon+O(\ln\epsilon^{-1})$ approximation algorithm for this problem, where d is the number of dimensions considered, while Bansal *et al.* presented in [16] a $\ln d + 1$ approximation algorithm. However all those algorithms are offline and still highly dependent on the number of dimensions.

Yet, interesting ideas can be explored: we can for example assume that memory can be considered as possibly taking a small number of different values (like $1GB$, $2GB$, $4GB$, ...). Such reasonable assumptions allow to obtain offline approximation algorithms [17] despite the negative results. An exciting line of research is to analyze which practical settings allow to derive provably efficient dynamic algorithms.

VI. Conclusion

In this paper, we present a novel way to address the problem of dynamic allocation of Virtual Machines onto Physical Machines in a Cloud Computing environment. We propose an algorithm which maintains an efficient allocation despite unpredictable variations of the CPU consumption of VMs over time, while keeping a good quality of service by ensuring that SLA violations are corrected (via migrations) as quickly as possible. More precisely, it ensures that at any point in time, the global CPU utilization of the platform is at least 66%, and that each time a PM becomes overloaded, at most 3 other bins are implied in the migrations necessary to repair the occuring SLA violations. Moreover all those migrations are independent from each other, and the maximum number of migrations is bounded by 6, which is strictly better than the algorithms already existing in the literature.

These strong theoretical guarantees strongly suggest that this algorithm would behave particularly well on actual scenarios. Average-case analysis of its performance thanks to simulation experiments would give very interesting insight about the robustness and feasibility of our approach. In particular, it would allow to find out how much load variability this algorithm can handle, as well as to explore the compromise between resource utilization and robustness. This would in turn provide a very interesting analysis of

the contexts in which it is feasible to use live migration to increase server consolidation. However, realistic execution traces (with information about load variation of individual VMs) of actual Cloud environments are difficult to obtain.

On a more algorithmic side, it would be very interesting to obtain the same kind of results in the multi-dimensional case (*i.e.* when taking into account memory constraint in addition to CPU consumption) and/or to the heterogeneous case (*i.e.* when we do not assume that the capacities of the PMs are all identical). Given the intrinsic difficulties of the underlying offline problems, these generalizations will probably require to identify relevant simplifying assumptions which are realistic in practice and allow to derive provably efficient solutions.

REFERENCES

[1] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, april 2010, pp. 27 –33.

[2] T. Vercauteren, P. Aggarwal, X. Wang, and T. Li, "Hierarchical forecasting of web server workload using sequential monte carlo training," *Signal Processing, IEEE Transactions on*, vol. 55, no. 4, pp. 1286–1297, 2007.

[3] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *Network and Service Management (CNSM), 2010 International Conference on*, oct. 2010, pp. 9 –16.

[4] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via vm multiplexing," in *Proceedings of the 7th international conference on Autonomic computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809052

[5] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of virtual machine live migration in clouds: A performance evaluation," *Cloud Computing*, pp. 254–265, 2009.

[6] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '09. New York, NY, USA: ACM, 2009, pp. 51–60. [Online]. Available: http://doi.acm.org/10.1145/1508293.1508301

[7] J. E. G. Coffman, M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: a survey," pp. 46–93, 1997.

[8] S. S. Seiden, "On the online bin packing problem," *Journal of the ACM*, vol. 49, p. 2002, 2001.

[9] E. Coffman Jr, C. János, and A. Zsbán, "Dynamic bin packing," in *SIAM J. COMPUT.* Citeseer, 1983.

[10] W. tat Chan, T. wah Lam, and P. W. H. Wong, "Dynamic bin packing of unit fractions items," in *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP*. Springer, 2005, pp. 614–626.

[11] J. Chan, P. Wong, and F. Yung, "On dynamic bin packing: An improved lower bound and resource augmentation analysis," *Algorithmica*, vol. 53, no. 2, pp. 172–206, 2009.

[12] Z. Ivković and E. Lloyd, "Fully dynamic algorithms for bin packing: Being (mostly) myopic helps," *Algorithms-ESA'93*, pp. 224–235, 1993.

[13] G. Gambosi, A. Postiglione, and M. Talamo, "Algorithms for the relaxed online bin-packing model," *SIAM journal on computing*, vol. 30, p. 1532, 2000.

[14] G. Woeginger, "There is no asymptotic ptas for two-dimensional vector packing," *Information Processing Letters*, vol. 64, no. 6, pp. 293–297, 1997.

[15] C. Chekuri and S. Khanna, "On multi-dimensional packing problems," in *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1999, pp. 185–194.

[16] N. Bansal, A. Caprara, and M. Sviridenko, "Improved approximation algorithms for multidimensional bin packing problems," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. IEEE, 2006, pp. 697–708.

[17] A. Caprara, H. Kellerer, and U. Pferschy, "Approximation schemes for ordered vector packing problems," *Naval Research Logistics (NRL)*, vol. 50, no. 1, pp. 58–69, 2003.

[18] L. Epstein and R. Van Stee, "Approximation schemes for packing splittable items with cardinality constraints," *Approximation and Online Algorithms*, pp. 232–245, 2008.