A branch-and-bound approach to virtual machine placement

Published in the *Proceedings of the 3rd HPI Cloud Symposium "Operating the Cloud"*, pages 49-63, 2015

Dávid Bartók
Budapest University of Technology and Economics
Department of Computer Science and Information Theory
Budapest, Hungary

Zoltán Ádám Mann
Budapest University of Technology and Economics
Department of Computer Science and Information Theory
Budapest, Hungary

***Abstract*—Finding the best mapping of virtual machines to physical machines in cloud data centers is a very important optimization problem, with huge impact on costs, application performance, and energy consumption. Although several algorithms have been suggested to solve this problem, most of them are either simple heuristics or use off-the-shelf, mostly integer linear programming (ILP) solvers. In this paper, we propose a new approach: a custom branch-and-bound algorithm that exploits problem-specific knowledge in order to improve effectiveness. As shown by empirical results, the new algorithm performs better than state-of-the-art general-purpose ILP solvers.**

I. Introduction

As cloud data centers (DCs) serve an ever-growing demand for computation, storage, and networking capacity, their operation is becoming a crucial issue. The energy consumption of DCs is of special importance because of both its environmental impact and its contribution to operational costs. According to a recent study, DC electricity consumption in the USA alone will increase to 140 billion kWh per year by 2020, costing US businesses 13 billion USD annually in electricity bills and emitting nearly 100 million tons of CO_2 per year [16].

In order to reduce energy consumption, DC operators use a combination of several techniques. Virtualization technology enables the safe co-existence of multiple applications packaged as virtual machines (VMs) on a single physical machine (PM), thus allowing high utilization of physical resources. Live migration makes it possible to move a working VM from one PM to another without noticeable downtime. Since the load of VMs fluctuates over time, this enables DC operators to flexibly react to such changes. In times of low demand, VMs can be consolidated to a low number of PMs, and the remaining PMs can be switched to a low-power state, leading to considerable energy savings. When load starts to rise, some PMs must be switched back to normal mode again so that VMs can be spread across a higher number of PMs.

Finding the best VM placement for the current load level is a tough optimization problem. First of all, multiple resource types must be taken into account, e.g., CPU, memory, disk, and network bandwidth. PMs have given capacity and VMs have given load along these dimensions, and this must be taken into account in VM placement. Moreover, the migration of VMs has a non-negligible overhead in the form of additional network traffic and additional load on the affected PMs. Thus, excessive migrations should be avoided.

In the past couple of years, several different approaches have been proposed for the VM placement problem. From an algorithmic point of view, these can be mostly grouped into two categories: (i) heuristics without any performance guarantees or theoretical underpinning and (ii) exact algorithms using off-the-shelf mathematic programming – mostly integer linear programming (ILP) – solvers [14]. It is dangerous to rely solely on heuristics because in some cases they can lead to extremely high costs or dramatic performance degradation of the involved applications [15]. On the other hand, the exact algorithms suggested so far all suffer from serious scalability issues, limiting their applicability to small problem instances.

In this paper, we propose a new approach, with the aim of finding a good compromise between practical applicability and theoretical soundness. Our approach is based on branch-and-bound, just like typical ILP solvers. However, in contrast to general-purpose ILP solvers, we can make use of problem-specific knowledge to make the search more effective. This is achieved by crafting customized procedures for controlling the branching behavior, custom bounding techniques etc.

II. Previous work

Several problem formulations have been suggested for the VM placement problem. They almost always include computational capacity of PMs and computational load of VMs. In fact, in many works, this is the only dimension that is considered [1]–[4], [6], [9], [10], [12], [22], [23]. Other authors included, beside the CPU, also some other resources like memory, I/O, storage, or network bandwidth [5], [7], [8], [21], [25].

Different objective or cost functions have been proposed. The number of active PMs is often considered because it largely determines the total energy consumption [3], [4], [6], [8], [23], [25]. Another important factor that some works considered is the cost of migration of VMs [6], [8], [20], [22].

Concerning the used algorithmic techniques, most previous works apply simple heuristics. These include packing algorithms inspired by results on the related bin-packing problem, such as First-Fit, Best-Fit, and similar algorithms [2]–[4], [9],

[11], [13], [22], [23], other greedy heuristics [17], [24] and straight-forward selection policies [1], [18], as well as metaheuristics [7], [8].

Some exact algorithms have also been suggested. Most of them use some form of mathematical programming to formulate the problem and then apply an off-the-shelf solver. Examples include integer linear programming [1] and its variants like binary integer programming [5], [13] and mixed integer non-linear programming [9]. Unfortunately, all these methods suffer from a scalability problem, limiting their applicability to small-scale problem instances.

III. Problem model

Let P denote the set of available PMs and V the set of VMs hosted in the DC. We consider d dimensions or resource types; e.g., if CPU capacity and memory are considered, then $d = 2$. The *capacity* of each PM and the *load* of each VM is a d-dimensional vector. For $p \in P$, its capacity is denoted by $cap(p) \in \mathbb{R}^d_+$, and for $v \in V$, its load is denoted by $load(v) \in \mathbb{R}^d_+$. Further, let $|P| = m$ and $|V| = n$.

The DC operator regularly re-optimizes the placement of the VMs in order to adapt to changes [20]. The current placement is given by $map_0 : V \to P$. Our aim is to determine a new mapping $map : V \to P$, subject to capacity constraints

$$\forall p \in P : \sum_{v: map(v)=p} load(v) \leq_d cap(p), \tag{1}$$

where $\leq_d$ is a relation between d-dimensional vectors; $(x_1, \ldots, x_d)^T \leq_d (y_1, \ldots, y_d)^T$ if and only if for each $1 \leq i \leq d$, $x_i \leq y_i$. map_0 may not satisfy the capacity constraints; even if it satisfied them at the time it was computed, the change in VM loads since then may have rendered it invalid.

A PM is *active* if it hosts at least one VM, i.e., $p \in P$ is active if $\exists v \in V, map(v) = p$. The number of active PMs is $act(map)$. Since energy consumption is largely determined by the number of active PMs, we should minimize $act(map)$.

A *migration* of $v \in V$ occurs if $map(v) \neq map_0(v)$. The number of migrations caused by map is given by $mig(map)$. Because of the overhead caused by migrations, we should minimize $mig(map)$ as well. We combine the two minimization objectives in a single cost function:

$$f(map) = \alpha \cdot act(map) + \mu \cdot mig(map), \tag{2}$$

where α and μ are given non-negative weights defining the relative importance of the two optimization goals. In addition, we require the number of migrations to be below a given limit:

$$mig(map) \leq K, \tag{3}$$

where K is a given non-negative number. This is sensible because too many migrations make the solution practically infeasible [19]; thus, mappings that would cause too many migrations must be excluded even if they lead to few active PMs and thus to good overall objective value.

To sum, our aim is to determine a new mapping map that minimizes (2), subject to constraints (1) and (3).

IV. Integer programming solution

As a baseline, we formulate the problem as an integer program and solve it with an off-the-shelf ILP solver.

Indexing VMs as v_i $(i = 1, \ldots, n)$ and PMs as p_j $(j = 1, \ldots, m)$, the following binary variables are introduced:

$$Alloc_{i,j} = \begin{cases} 1 & \text{if } v_i \text{ is allocated on } p_j \\ 0 & \text{otherwise} \end{cases}$$

$$Active_j = \begin{cases} 1 & \text{if } p_j \text{ is active} \\ 0 & \text{otherwise} \end{cases}$$

$$Migr_i = \begin{cases} 1 & \text{if } v_i \text{ is migrated} \\ 0 & \text{otherwise} \end{cases}$$

Using these variables, the integer program can be formulated as follows ($i = 1, \ldots, n$ and $j = 1, \ldots, m$):

$$\min \quad \alpha \cdot \sum_{j=1}^{m} Active_j + \mu \cdot \sum_{i=1}^{n} Migr_i \tag{4}$$

$$\text{s. t.} \quad \sum_{j=1}^{m} Alloc_{i,j} = 1 \qquad \forall i \tag{5}$$

$$Alloc_{i,j} \leq Active_j \qquad \forall i, j \tag{6}$$

$$\sum_{i=1}^{n} load(v_i) \cdot Alloc_{i,j} \leq_d cap(p_j) \qquad \forall j \tag{7}$$

$$Migr_i = 1 - Alloc_{i,map_0(v_i)} \qquad \forall i \tag{8}$$

$$\sum_{i=1}^{n} Migr_i \leq K \tag{9}$$

$$Alloc_{i,j}, Active_j, Migr_i \in \{0, 1\} \qquad \forall i, j \tag{10}$$

The objective function (4) is the same as before, consisting of the number of active PMs and the number of migrations. Equation (5) ensures that each VM is allocated to exactly one PM, whereas constraint (6) ensures that for a PM p_j to which at least one VM is allocated, $Active_j = 1$. Together with the objective function, this ensures that $Active_j = 1$ holds for *exactly* those PMs that accommodate at least one VM.

Constraint (7) is a straight-forward formulation of constraint (1) in terms of the binary variables $Alloc_{i,j}$. Equation (8) determines the values of the $Migr_i$ variables and constraint (9) corresponds to constraint (3).

V. Branch-and-bound algorithm

Our algorithm does not use the binary variables introduced for the ILP approach, but operates directly on the map function. It works with partial solutions, in which $map(v)$ is defined for a subset of the VMs, and traverses the space of partial solutions in a tree-like manner. For a partial solution, its children in the tree are obtained by selecting a VM that is not mapped yet and trying to map it to all PMs that have sufficient free capacity to host it: for each such PM, a different child partial solution is obtained.

The search starts with all VMs unmapped (the root of the tree), and goes down the tree by mapping one more VM in each step. If all VMs are mapped, then a solution has been found, corresponding to a leaf of the tree. The best solution that has been found so far (*best_so_far*), along with its cost (*best_cost_so_far*), is maintained throughout the algorithm. If the current branch of the search tree cannot be continued or there is no point in doing so, then the algorithm backtracks. This happens in the following cases:

- A leaf has been reached.
- The current partial solution has become infeasible, i.e.,
  - either there is a VM for which no PM has sufficient free capacity,
  - or the number of migrations exceeds the limit.
- All children of the current partial solution have been processed.
- The cost of any solution that extends the current partial solution is surely not lower than the cost of the best solution found so far.

In each of these cases, the algorithm backtracks by undoing the last VM mapping decision, i.e., going back to the parent node in the tree, essentially unallocating the last VM. Afterwards, the next child of the parent is tried, i.e., a new PM is selected for the unallocated VM. When the search would need to backtrack from the root, the algorithm terminates.

```
loop
    if all VMs mapped and cost $<$ best_cost_so_far
    then
        update best_so_far and best_cost_so_far;
    end
    if all VMs mapped or infeasible or all children
    visited or min_cost $\geq$ best_cost_so_far then
        // backtrack
        if we are in the root then
            return best_so_far
        end
        move back to parent;
    end
    if no VM selected yet then
        select VM;
    end
    move to next child;
end
```

Algorithm 1: Branch-and-bound procedure

The skeleton of the branch-and-bound procedure is shown in Algorithm 1. In the following, the non-trivial parts are described in more detail.

A. Incremental computations

During the algorithm, many details of the current partial solution are needed, e.g., its cost. Such characteristics can be simply computed directly from the partial solution itself. However, it is much more efficient to compute them incrementally. For example, we maintain the cost of the current partial solution in a variable, and whenever we go up or down in the tree, the necessary change is made to the stored cost value. This way, determining the cost of the current partial solution takes $O(1)$ steps instead of $O(n)$, which is an important difference as this is needed many times.

Beside the cost of the current partial solution, the following characteristics are maintained and incrementally updated:

- The number of migrations.
- The remaining free capacity of each PM.
- For each VM, the set of PMs that still have enough free capacity to host it.

B. VM selection

VMs can be selected in any order, but this order may have considerable impact on the running time of the algorithm. As the primary criterion for selecting the next VM, we use the first-fail principle, a common approach in constraint satisfaction algorithms: we select the VM with the lowest number of PMs that can host it. This helps to keep the number of children of the nodes of the tree (the branching factor) low and thus the whole tree relatively small.

There can be several VMs with the same number of possible hosting PMs, so we also apply a secondary strategy for tie-breaking: VMs with higher load are preferred. Just like in bin-packing, where sorting the items in decreasing order is known to improve the performance of packing algorithms, here it is also sensible to place the biggest VMs first.

In our case, VM loads are multi-dimensional, so it is not clear what is "bigger." We implemented multiple strategies for sorting d-dimensional vectors:

- Using the lexicographic order of the vectors
- According to the maximum of the dimensions (L^∞ norm)
- According to the sum of the dimensions (L^1 norm)

C. PM selection

After having selected a VM v, the PMs that can host it must be tried one after the other. Again, the order in which the PMs are tried may impact the performance of the algorithm.

One possibility is to sort the PMs according to their remaining capacity. Again, these are multi-dimensional vectors, so we implemented the same sorting strategies as for VMs, with the single difference that empty PMs are put at the end in order to foster better utilization of PMs that are already on.

Another idea is to start with the PM on which v resides according to map_0. Since this strategy only defines the PM that should be tried first, it can also be combined with any of the sorting strategies, which will then determine the order of the remaining PM candidates.

D. Lower bound on the cost

In Algorithm 1, min_cost denotes a lower bound on the cost of any solution that can arise as an extension of the current partial solution. If min_cost is not less than the best cost found so far, then we can backtrack from the current subtree. The question is how to compute a (non-trivial) lower bound.

Let us consider a partial solution, in which a subset $V_1 \subset V$ of the VMs have already been allocated to a subset $P_1 \subset P$ of the PMs. Let k_1 denote the number of migrations that been made when allocating the VMs of V_1. We have to allocate the remaining VMs of $V_2 := V \setminus V_1$ with at most $K' := K - k_1$

migrations. Ideally, we would like to find the minimal cost according to equation (2), given the constraints (1) and (3) and given the current partial allocation. This is a tough problem. Luckily, we just need a lower bound. This can be achieved by considering a relaxation of the problem: constraint (1) – the capacity constraints – will be disregarded.

The resulting problem is: given the current partial solution, what is the best cost in terms of the objective (2) that can be achieved by the allocation of V_2, if at most K' further migrations are allowed? A cost of $\alpha \cdot |P_1| + \mu \cdot k_1$ has already been incurred. For the remaining VMs, since the number of migrations is constrained, this limits how much the new mapping can differ from map_0.

Let C_0 be the cost of mapping each remaining VM as in map_0. If P_2 is the set of PMs in $P \setminus P_1$ that are used by map_0 for mapping V_2, i.e., $P_2 = \{p \in P \setminus P_1 : \exists v \in V_2 \; map_0(v) = p\}$, then $C_0 = \alpha \cdot (|P_1| + |P_2|) + \mu \cdot k_1$. For a mapping with lower costs, some PMs must be emptied, i.e., all their VMs migrated to other PMs. This decreases the cost if the number of VMs that have to be migrated is less than α/μ. In order to empty the maximum number of PMs, PMs with the least number of VMs should be emptied. Therefore, Algorithm 2 delivers optimal result for the relaxed problem.

```
foreach $p \in P_2$ do
    $a(p) := |\{v \in V_2 : map_0(v) = p\}|$;
end
sort $P_2$ in ascending order of $a(p)$;
$i = 1$;
$mig = 0$;
loop
    let $p$ be the $i$th element of $P_2$;
    if $a(p) \geq \alpha/\mu$ or $mig + a(p) > K'$ then
        return
    end
    // empty $p$
    $mig += a(p)$;
    $i$ ++;
    if $i > |P_2|$ then
        return
    end
end
```

Algorithm 2: Optimal solution for the relaxed problem

This can be simplified with the following ideas: (i) the actual mapping[1] delivered by the algorithm is not interesting, only its cost; (ii) the $a(p)$ values are typically small non-negative integers. For any non-negative integer j, let b_j denote the number of PMs in P_2 with $a(p) = j$, i.e., $b_j := |\{p \in P_2 : a(p) = j\}|$. Let J denote the highest j for which $b_j > 0$.

Algorithm 3 is the simplified version. We just have to iterate through the (j, b_j) numbers. For the first couple of j values, all b_j PMs with $a(p) = j$ can be emptied, followed

[1]What the algorithm returns is actually not a real mapping because it does not determine where to place the migrated VMs. Since the capacity constraints do not have to be observed now, this does not matter: they could be placed on any of the used PMs.

```
$cost = C_0$;
$mig = 0$;
$j = 0$;
while $j \leq J$ and $j < \alpha/\mu$ and $mig < K'$ do
    $t := \min(b_j, \lfloor (K' - mig)/j \rfloor)$;
    $mig += t \cdot j$;
    $cost -= (\alpha \cdot t - \mu \cdot t \cdot j)$;
    $j$ ++;
end
```

Algorithm 3: Simplified algorithm for the relaxed problem

by a case in which only some $t < b_j$ PMs can be emptied, resulting in $mig = K'$. Using Algorithm 3, the lower bound on the cost can be easily and quickly computed, if the values of C_0, J, and the b_j numbers are maintained.

E. Trading off running time and solution quality

All techniques so far help to reduce the running time of the algorithm on typical problem instances, without sacrificing optimality. However, the running time may still be too high for practical applicability. The following techniques reduce the running time further, but without guaranteeing optimality.

1) Symmetry breaking: In a real DC, it is common to have many PMs of the same type. As long as they do not host any VMs, their capacity is the same, introducing some symmetry in the problem. When looking for a host for the current VM v, of course the one on which map_0 maps v must be handled separately, but all others that have the same capacity are equivalent choices – at least "almost equivalent," as we will see. Hence, it suffices to try just one of them. For example, if there are 100 PMs, all of the same type, then two of them must be tried ($map_0(v)$ and one of the others) instead of 100, yielding a speedup of a factor 50.

Unfortunately, the PMs in question are not fully equivalent because they host different VMs initially (i.e., according to map_0). Placing v on one of the PMs may require one of the VMs that was initially on that PM to be migrated to another PM, whereas placing v on another PM may not lead to migrations, for example. By considering only one of these PMs for v, the search is not complete anymore: optimality is not guaranteed. Nevertheless, it can be a good heuristic.

2) Discarding small improvement possibilities: If we do not strive for optimality, then a sensible goal is to strive for a solution with cost at most γ times the optimum, where $\gamma > 1$ is some given constant. Recall from Algorithm 1 that we backtrack if $min\_cost \geq best\_cost\_so\_far$. Now, this condition can be changed to $min\_cost \geq best\_cost\_so\_far/\gamma$, resulting in more aggressive pruning. The justification is that either $best\_cost\_so\_far$ is already within γ times the optimum, in which case we do not need any further search, or otherwise the condition $min\_cost \geq best\_cost\_so\_far/\gamma$ implies that min_cost is higher than the optimum, so that pruning this part of the search tree does not remove the optimum.

3) Limiting the runtime: The most drastic measure is to simply stop the search after some given time limit, and return the best allocation found so far.

F. Further remarks

The algorithm can be easily extended to accommodate further constraints in the form of additional pruning rules. E.g., colocation or anti-colocation requirements may exist for certain sets of VMs. These can be ensured by removing the non-compliant options from the list of PM candidates for each VM. E.g., if VMs v_1 and v_2 must not be colocated and the algorithm decides to map v_1 to PM p, then p must be removed from the list of possible PMs of v_2.

The algorithm works with an arbitrary number of dimensions d. Considering the impact on the running time, the steps of the algorithm are either agnostic of the value of d, or have a linear runtime in d. Further, d is small in practice. Thus, there is no combinatorial explosion with respect to d.

VI. Evaluation

We compare, by means of simulation experiments, three algorithms. The first two methods use off-the-shelf ILP solvers on the ILP formulation of Section IV, as suggested so far in the literature. The solvers are: lp_solve[2] 5.5.2, one of the leading free open-source packages and Gurobi[3] 6.0.5, a successful commercial product. The third method is our branch-and-bound (BB) algorithm. All measurements were carried out on a desktop PC with 2.6 GHz Pentium E5300 Dual-Core CPU and 3 GB DDR2 800MHz RAM, running MS Windows 7.

Problem instances were generated in the following way. The number of dimensions, d, is set to 2. There are 4 PM types; each PM belongs to a randomly selected PM type. In each dimension, the capacity of PM types is randomly generated between 8 and 14, whereas the load of each VM is randomly taken between 1 and 5. The number of PMs and VMs was determined for each experiment separately (see below), varying between 25 and 4000. The number of allowed migrations, K, is set to 10% of the number of PMs. The weights in the cost function are $\alpha = 10$ and $\mu = 1$.

The initial allocation of VMs to PMs is generated in two steps. First, each VM is randomly mapped on one of the PMs. Such a random mapping may lead to an unrealistically high number of overloaded PMs that cannot be repaired with the limited number of migrations. Hence, in a second step, the First-Fit heuristic is used to pack the VMs into the PMs, with the extension that VMs that did not fit into any PM remain on the PM determined by the random placement. The result is a mapping that has likely few overloaded PMs and some room for consolidation; hence, it models well the typical initial mapping for a VM placement re-optimization algorithm.

Each algorithm is run on each problem instance with a timeout of 60 seconds. All presented numbers are the median of 10 measurements. Moreover, we also present lower bounds for the optimum; these were obtained by applying the bounding method of Section V-D before any branching has taken place.

[2]http://lpsolve.sourceforge.net/5.5/

[3]http://www.gurobi.com/

TABLE I
Configuration of the branch-and-bound algorithm

Technique	Used variant
VM selection	First-fail; tie-breaking: sorting according to the maximal dimension
PM selection	Initial PM first; rest sorted lexicographically
Lower bound	Used as described
Symmetry breaking	Used as described
Pruning small improvements	Not used ($\gamma = 1$)

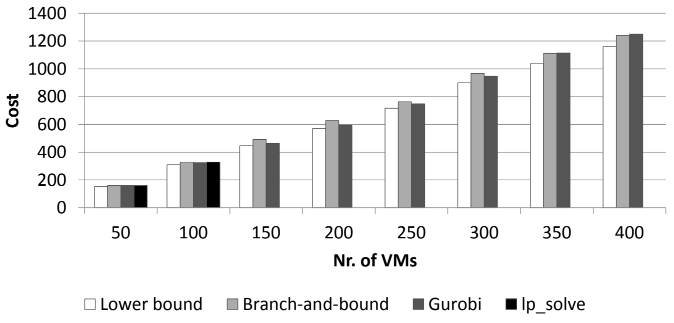

Fig. 1. Scalability results on small problem instances

A. Parameter tuning

First, we aimed at finding good settings for the parameters of the BB algorithm. We used randomly generated problem instances with m varying between 25 and 450 and n varying between 50 and 900. Most of the techniques built into the algorithm proved to be indeed useful. The only exception was the technique described in Section V-E2 to cut off branches with small improvement possibilities. The reason why this did not help is probably that – as shown below – the algorithm quickly finds solutions that are quite near the optimum, so that only small improvements are possible afterwards.

The configuration that turned out to be best and was used for the later experiments is shown in Table I.

B. Comparison

Our main objective was to assess the scalability of the three algorithms. To that end, we considered problem instances of increasing size. For this experiment, we fixed the ratio of VMs to PMs to 2, and increased the number of PMs from 25 to 2000, with the number of VMs ranging from 50 to 4000.

To enhance visibility, the results are split into two figures. Fig. 1 shows results for problem instances with $n \leq 400$, whereas Fig. 2 shows the results for bigger problem instances.

In Fig. 1, all algorithms perform very similarly for the smallest problem instances. For $n \geq 150$, lp_solve fails to deliver a solutio. The other two algorithms continue to deliver solutions, with Gurobi performing slightly better than BB for $150 \leq n \leq 300$. However, BB closes in on Gurobi at around $n = 350 \ldots 400$. In Fig. 2 we see that for $n \geq 600$, BB already consistently outperforms Gurobi, with the latter increasingly drifting away from the optimum. After

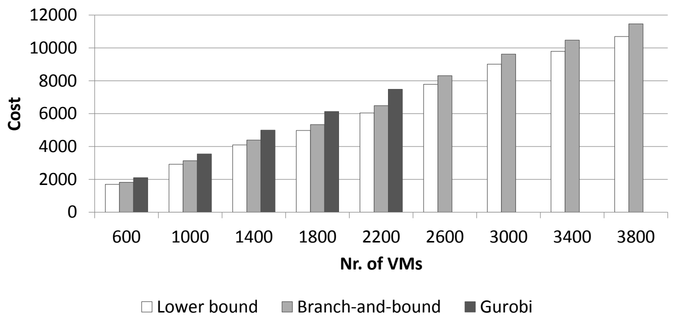

Fig. 2. Scalability results on big problem instances

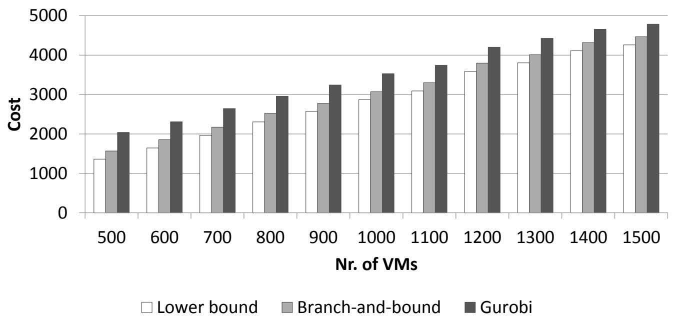

Fig. 3. Instances with different density ($m = 500$ is constant)

$n \geq 2600$, Gurobi fails to find a valid solution within the given time limit. BB on the other hand continues to deliver results.

The quality of the results found by BB is excellent: they are in most cases within 10% of the lower bound, and therefore, also within 10% of the optimum.

Finally, we assessed the effect of the load density (the n/m ratio). With 500 PMs, we varied the number of VMs from 500 (lightly loaded DC) to 1500 (highly loaded DC). As can be seen in Fig. 3, BB consistently outperforms Gurobi for all densities (lp_solve did not produce valid results in this range).

In our future work, we plan to undertake a more detailed empirical analysis of the algorithm's performance, also comparing it with other algorithms on more realistic test data. Unfortunately, lacking generally accepted benchmarks, this must be done in an ad-hoc manner.

Acknowledgment

This work was partially supported by the Hungarian Scientific Research Fund (Grant Nr. OTKA 108947).

References

[1] D. M. Batista, N. L. S. da Fonseca, and F. K. Miyazawa. A set of schedulers for grid networks. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC'07)*, pages 209–213, 2007.

[2] A. Beloglazov, J. Abawajy, and R. Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28:755–768, 2012.

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

[4] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing SLA violations. In *10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128, 2007.

[5] R. v. d. Bossche, K. Vanmechelen, and J. Broeckhove. Cost-optimal scheduling in hybrid IaaS clouds for deadline constrained workloads. In *IEEE 3rd International Conference on Cloud Computing*, pages 228–235, 2010.

[6] D. Breitgand and A. Epstein. SLA-aware placement of multi-virtual machine elastic services in compute clouds. In *12th IFIP/IEEE International Symposium on Integrated Network Management*, pages 161–168, 2011.

[7] Y. Gao, H. Guan, Z. Qi, Y. Hou, and L. Liu. A multi-objective ant colony system algorithm for virtual machine placement in cloud computing. *Journal of Computer and System Sciences*, 79:1230–1242, 2013.

[8] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Resource pool management: Reactive versus proactive or let's be friends. *Computer Networks*, 53(17):2905–2922, 2009.

[9] M. Guazzone, C. Anglano, and M. Canonico. Exploiting VM migration for the automated power and performance management of green cloud computing systems. In *1st International Workshop on Energy Efficient Data Centers*, pages 81–92. Springer, 2012.

[10] G. Jung, M. A. Hiltunen, K. R. Joshi, R. D. Schlichting, and C. Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *IEEE 30th International Conference on Distributed Computing Systems*, pages 62–73, 2010.

[11] A. Khosravi, S. K. Garg, and R. Buyya. Energy and carbon-efficient placement of virtual machines in distributed cloud data centers. In *Euro-Par 2013*, pages 317–328. Springer, 2013.

[12] D. G. do Lago, E. R. M. Madeira, and L. F. Bittencourt. Power-aware virtual machine scheduling on clouds using active cooling control and DVFS. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, 2011.

[13] W. Li, J. Tordsson, and E. Elmroth. Virtual machine placement for predictable and time-constrained peak loads. In *Proceedings of the 8th International Conference on Economics of Grids, Clouds, Systems, and Services (GECON 2011)*, pages 120–134. Springer, 2011.

[14] Z. Á. Mann. Allocation of virtual machines in cloud data centers – a survey of problem models and optimization algorithms. *ACM Computing Surveys*, 48(1), 2015.

[15] Z. Á. Mann. Rigorous results on the effectiveness of some heuristics for the consolidation of virtual machines in a cloud data center. *Future Generation Computer Systems*, 51:1–6, 2015.

[16] Natural Resources Defense Council. Scaling up energy efficiency across the data center industry: Evaluating key drivers and barriers. http://www.nrdc.org/energy/files/data-center-efficiency-assessment-IP.pdf, 2014.

[17] M. A. Salehi, P. R. Krishna, K. S. Deepak, and R. Buyya. Preemption-aware energy management in virtualized data centers. In *5th International Conference on Cloud Computing*, pages 844–851. IEEE, 2012.

[18] L. Shi, J. Furlong, and R. Wang. Empirical evaluation of vector bin packing algorithms for energy efficient data centers. In *IEEE Symposium on Computers and Communications*, pages 9–15, 2013.

[19] W. Song, Z. Xiao, Q. Chen, and H. Luo. Adaptive resource provisioning for the cloud using online bin packing. *IEEE Transactions on Computers*, 63(11):2647–2660, 2014.

[20] P. Svärd, W. Li, E. Wadbro, J. Tordsson, and E. Elmroth. Continuous datacenter consolidation. Technical report, Umea University, 2014.

[21] L. Tomás and J. Tordsson. An autonomic approach to risk-aware data center overbooking. *IEEE Transactions on Cloud Computing*, 2(3):292–305, 2014.

[22] A. Verma, P. Ahuja, and A. Neogi. pMapper: power and migration cost aware application placement in virtualized systems. In *Middleware 2008*, pages 243–264, 2008.

[23] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 USENIX Annual Technical Conference*, pages 355–368, 2009.

[24] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, 2009.

[25] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. G., T. Christian, and L. Cherkasova. 1000 islands: an integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12(1):45–57, 2009.