Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics

Allocating virtual machines in a cloud environment

Dávid Bartók
Project Laboratory

Advisor: **Dr. Zoltán Ádám Mann**
Department of Computer Science and Information Theory

May 5, 2015

Contents

1 Introduction

Cloud computing is becoming more and more widespread nowadays. The reasons for this phenomenon lie in the characteristics of the cloud - it allows the user to easily retain a flexible and scalable IT infrastructure for various purposes. The user does not have to be concerned about the operation and maintenance of the physical hardware, because they can use virtual machines granted by the cloud provider. These virtual resources can usually be accessed through the Internet, are conveniently resizable and may follow a pay-per-use model.

The cloud provider uses a shared infrastructure, dividing its available physical resources among its clients. Simply put, this means that a physical machine (PM) of the provider can serve as a host for more than one virtual machine (VM), possibly originating from multiple clients.

Optimization of operation costs is certainly crucial to the cloud providers [1]. That is the reason why the Virtual Machine Assignment (VMA) problem has gained significant attention in the last few years. In this problem, we assign the VMs requested by the clients to the PMs available at the provider. The goal is to find the assignment which yields the lowest cost.

The cost can incorporate the load of the physical machines and the number of VM migrations required to achieve the allocation. This is due to the fact that when we create a new allocation, some of the VMs might have to be reassigned (migrated) from their previous PM to the newly designated one.

Several other factors can also be taken into account when computing the cost, e.g., the communication costs between VMs.

2 Problem formulation

The VMA problem can be formulated in several ways, depending on the cloud model and the aspects taken into account when trying to determine the cost of an allocation.

The simplest formulation of the problem would be as follows:

Input:

- n VMs having resource demand $c_1, c_2, ..., c_n$
- m PMs, each having resource capacity C

Output:

- Assignment of VMs to PMs
- Capacity must not be exceeded for any PM
- Number of used PMs should be minimized

The above formulation is essentially equivalent to the bin-packing problem, which is NP-hard. However, efficient approximation algorithms exist which can yield results close to the optimum in polynomial time [2]. In my project I used an extended variant of the problem [3]:

Input:

- n VMs having demand for r resource types $c_{i,k} (i = 1, ..., n; \quad k = 1, ..., r)$
- m PMs having capacity for r resource types $C_{j,k} (i = 1, ..., m; \quad k = 1, ..., r)$
- Initial assignment $s : \{1, ..., n\} \rightarrow \{1, ..., m\}$

Output:

- Assignment of VMs to PMs
- Capacity must not be exceeded for any resource of any PM
- Weighted sum of the amount of PMs used and number of migrations should be minimized

The latter version of the problem makes it possible to define different types of PMs with multiple resources (CPU, RAM, Hard-drive etc.). The number of migrations required is also taken into account when determining the cost of an allocation. This problem, although still containing several simplifications, is much more realistic than the one-dimensional bin-packing variant. A two-dimensional version of this problem is depicted in Figure 1, with CPU and RAM as the two resource types. Three VMs are already allocated to a PM, and its RAM capacity is almost exhausted.

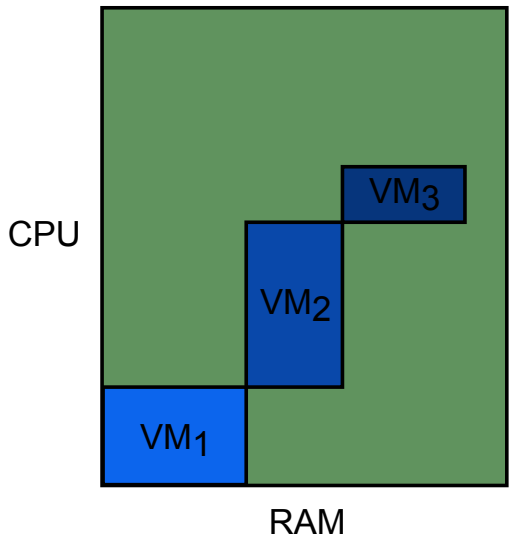

Figure 1: Packing VMs into a PM

3 The software

This section presents the software created in the project. I have written all code in C++ and compiled it using Visual C++ 2013.

3.1 Algorithms

The first algorithm I implemented was a recursive branch and bound. This is illustrated with pseudo-code in Listing 1. We start at the first VM, and try out all possible PM candidates for this VM in a loop. If an invalid allocation is created, we deallocate the current VM and continue with the next PM candidate. In case the allocation is valid, its cost is computed. If this is higher than the best solution found so far, we once again have to deallocate the current VM and continue with the next PM candidate. At this point we know that the cost is lower than that of our best allocation so far. This means that if all VMs are allocated, we can update our minimal cost to the current one. In the event that not all VMs are assigned, we move down the search tree and try to allocate the next VM. At the end of the loop we return from this function, meaning that the current branch is exhausted. Before returning we have to deallocate the current VM.

```
solveRecursive(VMHandled)
{
    PMCandidates = getPMCandidates();
    for (candidate in PMCandidates)
        allocate(VMHandled, candidate); // allocate VM to candidate PM
        if (not isAllocationValid())
            deAllocate(VMHandled);
            continue;

        cost = computeCost();
        if (cost >= bestCostSoFar) // bound
            deAllocate(VMHandled);
            continue;
        if (allVMsAllocated()) // new best allocation
            bestCostSoFar = cost;
            bestAllocationSoFar = allocation;
            deAllocate(VMHandled);
            continue;
        else
            solveRecursive(getNextVM(VMHandled));

        deAllocate(VMHandled);
}
```

Listing 1: Recursive allocation algorithm

Although this implementation is fairly simple, it leaves much to be desired. Because of the recursiveness, backtracking is implemented implicitly, by returning from the function to the previous VM. This makes any kind of backtrack optimization (e.g., dynamically computing some of the variables) very hard to implement. In addition to this, recursive backtrack algorithms are often less effective than their iterative counterparts because of the large call stack they produce. Due to these reasons, I also implemented an iterative algorithm, and carried out all further work with this version. The iterative algorithm is described in Listing 2.

There are several differences to the recursive implementation. Naturally, the function doesn't call itself, but instead runs in a loop. At the start of this loop we check if the current branch is exhausted. If this is the case, we backtrack to the previous VM. Here we have to check if there are still any more valid possibilities and if not, the algorithm is finished. If there are still some possibilities left, we deallocate the current VM (the present state is just after the backtracking, so no more allocations have to be checked in which this VM receives its current value). Computing the cost, eventually bounding the search space and updating the best solution found so far work just like in the recursive approach. At the end of the function, we continue the loop instead of recursively calling the function.

Backtracking is implemented by saving every allocation in a stack, and when an assignment has to be undone, we can easily retrieve which VM was allocated in the latest assignment and deallocate it.

One can also notice that here there is no validity checking for the allocations. This is because the invalid allocations are not investigated in this algorithm. The method for this is explained in more detail in Section 3.3. Both the recursive and iterative implementations are deterministic algorithms and they always return the optimal results to the problems.

3.2 Structure

The structure used by the program is depicted in Figure 2. The **VMAllocator** class is responsible for carrying out the allocation algorithm. It has a corresponding parameter setup in **AllocatorParams**, according to which the allocator's behavior differs. The VMA problems are stored in a separate **AllocationProblem** class and

```
solveIterative()
{
    while(true)
        if (currentBranchExhausted())
            VMHandled = backtrackToPreviousVM();
            if (allPossibilitiesExhausted())
                break;
            deAllocate(VMHandled);

        PMCandidate = getNextPMCandidate(VMHandled);
        allocate(VMHandled, PMCandidate);

        cost = computeCost();
        if (cost >= bestCostSoFar)
            deAllocate(VMHandled);
            continue;
        if (allVMsAllocated())
            bestCostSoFar = cost;
            bestAllocationSoFar = allocation;
            deAllocate(VMHandled);
            continue;
        else
            VMHandled = getNextVM();
}
```

Listing 2: Iterative allocation algorithm

consist of **PM**s and **VM**s. Problems can be created using the **ProblemGenerator**, which can be customized to produce instances with specific characteristics.

The generator can also read VMA problems from files. The files have a simple input format: the first line has to contain the number of VMs (n) and PMs (m). Then n lines follow, each containing the description of one VM, first listing the resource demands then the index of the initially assigned PM. For an r-dimensional problem, this results in $r + 1$ numbers. In the last m lines of the file, the PMs have to be defined. This happens by simply listing their capacities in each dimension (each line here has r numbers). Below is an example of a valid input file with two resource dimensions:

```
3 2

6 3 0
4 2 1
2 2 0

14 8
6 5
```

The **Change** class represents changes during the execution of the algorithm and can be used for backtracking optimization (for saving information about the state of the allocator class).

During the course of the algorithm, VMs and PMs are identified by their index in the **AllocationProblem**. However, we are free to change the order of the items in the problem, because the assignments to PMs are done according to a unique and consistent ID.

The structure was designed to be flexible enough to allow for optimization of the algorithm. Consequently, functions in the allocator class are well separated, and can be individually modified without

affecting each other, as long as they conform to their interfaces. The modifications described in the following section had no effect on the class diagram whatsoever (apart from adding the parameters to the **AllocatorParams** class of course).

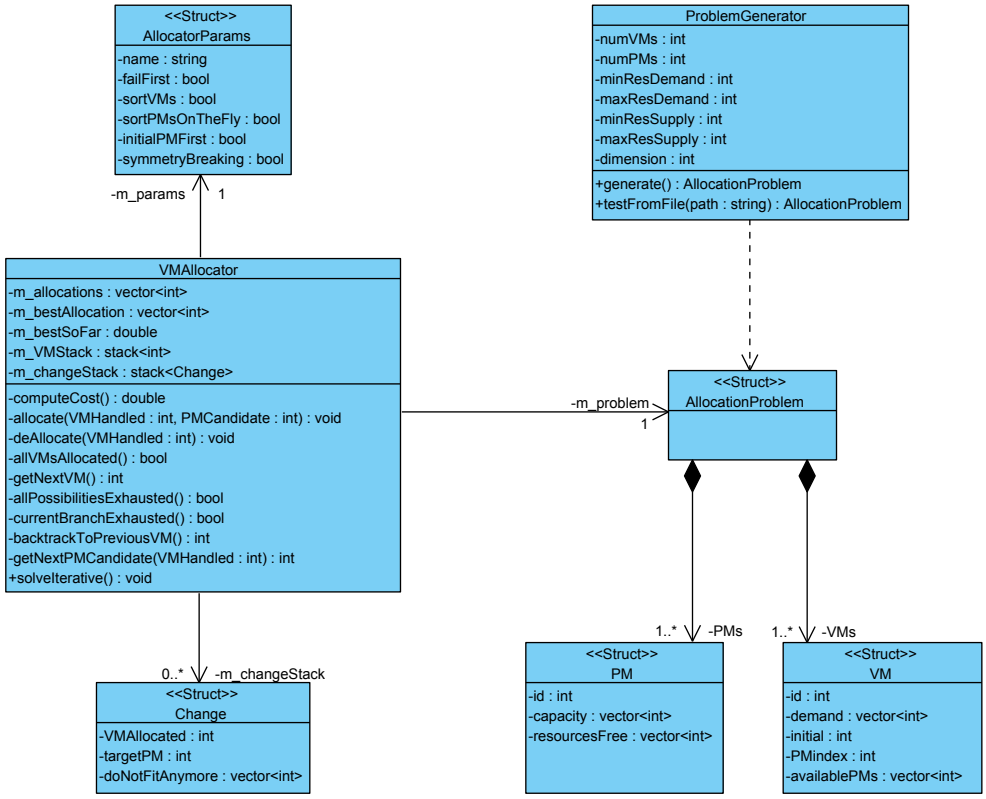

Figure 2: Class diagram

3.3 Optimizations

The basic algorithm described in Listing 2 can be optimized in several ways. Actually, as mentioned above, Listing 2 already contains an important optimization in itself: we only check the valid allocations. This is achieved by keeping track of possible PMs for every VM (where the PM has at least as much resources free in each dimension as demanded by the VM). The PM candidates are extracted directly from this list, thus making sure that every candidate is a valid one.

This list of course has to be maintained, because during allocations and deallocations the possible PMs for each VM can change. More precisely, after every allocation exactly one PM (the target of the assignment) might not be a valid candidate anymore for some VMs. Consequently, it is enough check at each allocation for every VM whether they still fit in this target PM or not. The VMs that do not fit anymore can be saved, making the modification to the lists easily undoable at the deallocation. This is outlined in the **Change** class of Figure 2.

The list defined above makes it possible to always choose the VM with the least amount of available PMs to process next. This is called the *fail first* heuristic, which is done in the hope of quickly narrowing the search space down. This heuristic doesn't affect the optimality of the algorithm.

Apart from this method, some other optimization heuristics are also implemented. These can be enabled or disabled through the **AllocatorParams** class:

- **sortVMs**: Sorts the VMs at the start of the algorithm, meaning that if two VMs fit on exactly the same amount of PMs, we allocate the larger one first. The reason for this would be that allocating the largest VM might narrow the search space down the most.

- **sortPMsOnTheFly**: Sort available PM list in ascending order before iterating through the possibilities. This means that we try the PM candidate with the least remaining resources first, assuming that the VM fits there the best.
- **initialPMFirst**: We always try the PM first, which was the initial assignment of the current VM (naturally only if this still results in a valid allocation). This is done in the hope that not migrating a VM might lead to the lowest total cost.
- **symmetryBreaking**: If we have several empty PMs with the same amount of capacities, we only try to allocate a VM on one of them, afterwards we skip over the rest. Although the above heuristics only change the order in which the VMs/PMs are visited, this one actually leaves some possibilities completely unexplored. Although empty PMs with the same capacities are here considered to be the same, often they are not, because the PMs can have different VMs allocated to them initially. Therefore, using symmetry breaking causes the algorithm to lose its optimality.

Please note that in sorting the VMs and PMs, currently lexicographical order is used. Some other ideas for defining an ordering among the machines are outlined in Section 5.

4 Results

In this part, I present some tests to evaluate the performance of the various heuristics introduced in the previous section. All experiments were performed on a PC with a 2,6 GHz Pentium E5300 Dual-Core CPU, 3 GB DDR2 800 MHz RAM under Windows 7.

In a first test I have evaluated the performance of the fail first heuristic on 50 problems, each with 14 VMs and 7 PMs (chosen out of two types). All problems had two resource dimensions, with VM resource demands being integers (inclusively) between 1 and 5, PM capacities similarly integers between 5 and 10. I used two configurations of the algorithm: one with the fail first heuristic, and one just iterating through the VMs sequentially. The **sortVMs** option, however, was enabled in both cases, meaning that the "simple" algorithm examined the VMs in descending order. The results can be seen in Figure 3. The graph clearly shows that the version using the fail first heuristic is much faster than the simple one.

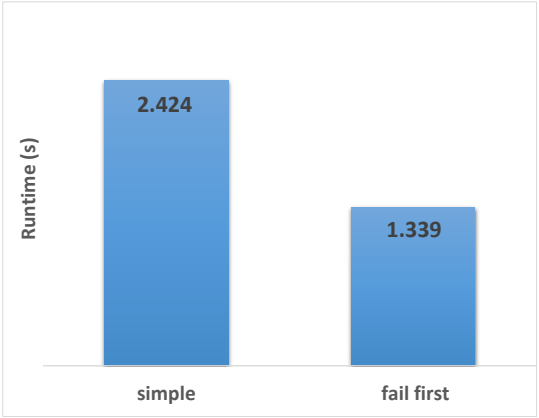

Figure 3: Evaluating the fail first heuristic

In a second test I compared the performance of five configurations on relatively small problems, all using the fail first heuristic and sorting the VMs at the start of the algorithm. One version had no additional optimizations, the second one had the **sortPMsOnTheFly** and the third one the **initialPMFirst** flag set. I also examined two symmetry breaking variants, one with and one without the **initialPMFirst** option. The test consisted of 100 randomly generated instances, each with 12 VMs, 6 PMs (once again chosen out of two types). The problems were two-dimensional, with VM resource demands being either 4 or 5, and PM capacities either 9 or 10. The results can be seen in Figure 4.

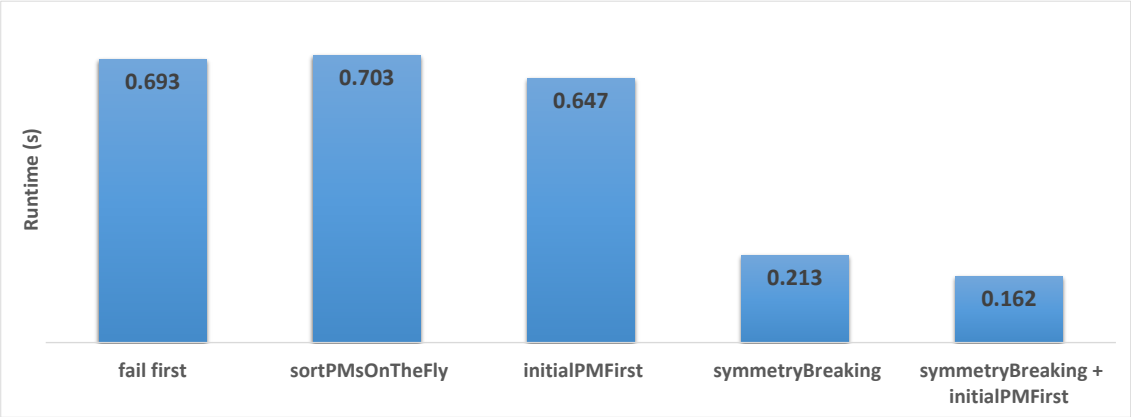

Figure 4: Comparing runtimes on small problems

One can immediately notice that the symmetry breaking versions are more than three times faster than the algorithms lacking this heuristic. An interesting piece of information, which cannot be seen in the graph, is that symmetry breaking yielded on average only 0.9% worse solutions than the optimum.

Another intriguing thing to note is that sorting the PMs on the fly does not really have an effect on performance. Also, in case of the optimal algorithms, choosing the initial PM first causes a speedup of only about 7%. However, if this method is used in conjunction with symmetry breaking, the speedup reaches about 24% (compared to the original symmetry breaking variant).

In a final test, I have used the same configurations as before, though on slightly larger problems. The test had 50 problems, with exactly the same parameters as in the first test (14 VMs, 7 PMs, resource ranges 1-5 for the VMs and 5-10 for the PMs). This generally lead to a larger runtime and more unsolvable problems. The results can be seen in Figure 5.

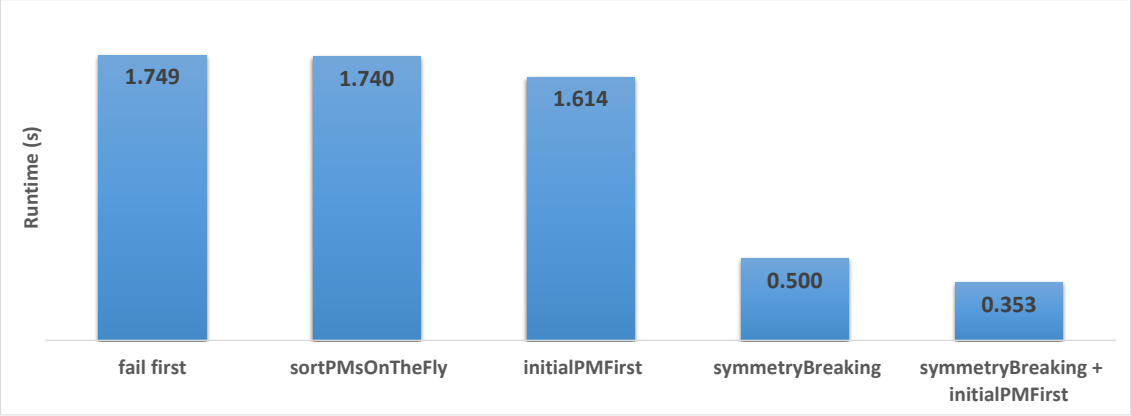

Figure 5: Comparing runtimes on larger problems having more variety

The runtime distribution is very similar to that of the first test, with symmetry breaking once again performing more than three times as fast as the algorithms giving optimal results. Similarly to the first test, symmetry breaking gives on average about 1% worse solutions than the optimum.

5 Summary and future work

In this document, I have presented my work in creating a virtual machine allocator program for cloud systems. I first clarified what the VMA problem is and why it is of great importance. Afterwards, I formally defined the problem model I adopted in my work. Then, I explained the algorithms implemented,

the structure used and some optimizations utilized. Finally, I demonstrated the results of my work by comparing different parametrizations of the algorithm.

We have seen that even very small problems take relatively long time to solve, due to the NP-hard nature of the VMA problem. Heuristics can speed up the process to a degree, although some can cause the loss of optimality. In future work, it could be examined which part of the algorithm takes up the most time by means of profiling, and targeted optimizations could be performed on this component.

Another direction would be to further explore sorting VMs and PMs - as mentioned before, currently lexicographical order is used. However, this doesn't necessarily describe the relationship between the machines well. Maybe a sum or maximum of resources across all dimensions would work better in this case.

There are also countless other directions one could consider when researching this problem, ranging from parallelization of the algorithm, to including some learning techniques.

References

[1] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing As the 5th Utility. *Future Gener. Comput. Syst.*, 25(6):599–616, June 2009.

[2] Edward G. Coffman, Gabor Galambos, Silvano Martello, and Daniele Vigo. Bin Packing Approximation Algorithms: Combinatorial Analysis. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 151–207. Springer US, 1999.

[3] Zoltán Ádám Mann. Modeling the virtual machine allocation problem. In *Proceedings of the International Conference on Mathematical Methods, Mathematical Models and Simulation in Science and Engineering (MMSSE 2015)*, pages 102–106, 2015.