Multicore-aware virtual machine placement in cloud data centers

Zoltán Ádám Mann

Published in *IEEE Transactions on Computers*, volume 65, number 11, pages 3357-3369, 2016

Abstract—Finding the best way to map virtual machines (VMs) to physical machines (PMs) in a cloud data center is an important optimization problem, with significant impact on costs, performance, and energy consumption. In most situations, the computational capacity of PMs and the computational load of VMs are a vital aspect to consider in the VM-to-PM mapping. Previous work modeled computational capacity and load as one-dimensional quantities. However, today's PMs have multiple processor cores, all of which can be shared by cores of multiple multicore VMs, leading to complex scheduling issues within a single PM, which the one-dimensional problem formulation cannot capture. In this paper, we argue that at least a simplified model of these scheduling issues should be taken into account during VM placement. We show how constraint programming techniques can be used to solve this problem, leading to significant improvement over non-multicore-aware VM placement. Several ways are presented to hybridize an exact constraint solver with common packing heuristics to derive an effective and scalable algorithm.

Index Terms—VM placement; VM consolidation; cloud computing; data center; optimization algorithms; constraint programming

1 Introduction

As cloud computing is becoming mainstream, cloud data centers (DCs) must serve an ever-growing demand for computation and storage capacity. As a result, the operation of DCs is becoming ever more challenging: infrastructure providers must find the right balance between the conflicting aims of keeping costs down, reducing energy consumption, and adhering to Service Level Agreements (SLAs) on the availability and performance of the hosted applications [13].

A key driving force in cloud adoption is the proliferation of virtualization technologies, allowing the secure and – more or less – isolated co-existence of multiple virtual machines (VMs) on the same physical machine (PM). This in turn enables a healthy utilization of physical resources. Live migration, the ability to move a VM from one PM to the other with practically no downtime [40], makes it possible to continuously adapt the VM-to-PM mapping to changes in the VMs' load and the PMs' availability [41].

As a consequence of its business drivers and the technological possibilities, an infrastructure provider will regularly re-optimize the VM-to-PM mapping in its DC, with the aim of consolidating the VMs to the minimal number of PMs that can accommodate them without breaching the quality goals laid down in the SLA, and switch off the PMs that were freed up in order to save energy [39]. Determining the best VM-to-PM mapping is the *VM placement problem*.

Because of its importance and inherent difficulty, a huge number of approaches have been proposed to model and solve this problem. However, as shown by our recent survey, the state of the art in VM placement research is still unsatisfactory concerning both the used problem models and algorithms [31]. Most previous works agree that the computational capacity of PMs and the computational load of VMs are crucial to take into account in VM consolidation. However, computational capacity and computational load are almost always captured by a single number per machine, turning VM placement into a simplistic one-dimensional problem, in which a set of VMs can be placed on a PM if and only if the sum of their CPU loads does not exceed the CPU capacity of the PM.

In reality, both PMs and VMs may have multiple CPU cores. When a VM is mapped to a PM, each of the VM's CPU cores (vCPUs) must also be mapped to one of the PM's CPU cores (pCPUs); a pCPU can be shared by multiple vCPUs. Therefore, the question whether a set of VMs can be mapped to a PM is actually a more difficult one, involving a non-trivial scheduling problem. This scheduling problem is solved at runtime by the hypervisor scheduler. Existing algorithms for VM placement ignore this problem by considering PM and VM CPUs as a whole and not looking into their components.

The main thesis of this paper is that *ignoring the scheduling of cores during VM placement is an over-simplification that may lead to suboptimal VM placement*. This issue will be discussed in detail in Section 3, but for the moment, it will be illustrated with an example.

Consider a quad-core PM with 4000 MIPS (million

• *The author is with Budapest University of Technology and Economics*

instructions per second[1]) available capacity per core. (This is the capacity available to VMs, after subtracting the load of the VM Manager and other system software.) Further, consider 3 dual-core VMs with the following load:

- VM 1: core 1 – 2200 MIPS, core 2 – 2100 MIPS
- VM 2: core 1 – 2100 MIPS, core 2 – 2000 MIPS
- VM 3: core 1 – 2000 MIPS, core 2 – 1800 MIPS

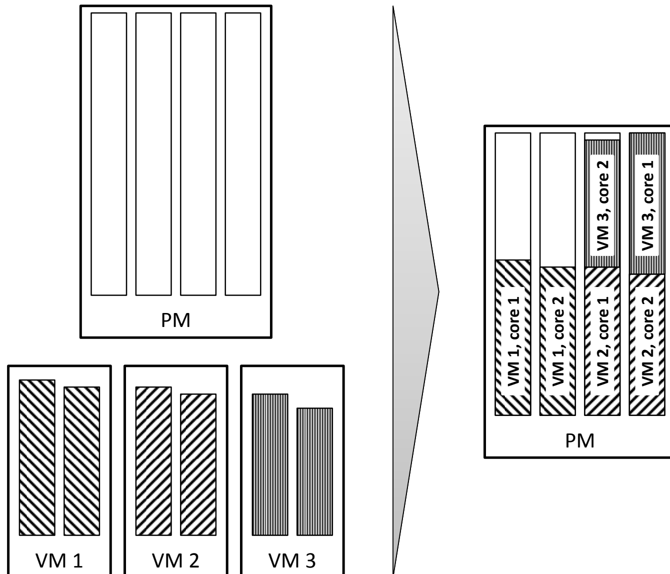


Fig. 1. Example mapping of VM cores on PM cores

As shown in Fig. 1, it is possible to map all vCPUs on the pCPUs. However, if the load of core 1 of VM 3 is slightly increased, for instance to 2100 MIPS, then such a mapping does not exist anymore. This can be seen easily since the loads of the cores of VM 1 and 2 are such that no two of them can be mapped on the same pCPU, so that they will occupy at least 2000 MIPS from each pCPU, and therefore, no pCPU will have sufficient remaining capacity for vCPU 1 of VM 3.

Some previous works suggested to model a multicore processor with k cores and c capacity per core as a single-core processor with capacity $k \cdot c$ [6]. However, this approximation can be quite imprecise. In the above example, the total capacity of the PM is 16,000 MIPS, the total load of the VMs (when the load of the first core of VM 3 is increased to 2100 MIPS) is 12,300 MIPS, way below the capacity of the PM. Yet, as we have seen, the PM cannot satisfy the computational requirements of the three VMs. In other words, if a VM placement algorithm looks only at the total capacity of PMs and total load of VMs, without considering the mapping of vCPUs on pCPUs, it may consolidate the three VMs onto the PM, leading to an overload of one of the pCPUs and thus potentially to an SLA violation for the affected VMs.

1. Throughout the paper, MIPS is used as the unit of CPU capacity and CPU load. However, this is only an example. Other units could also be used, e.g., clock cycles, FLOPS, or some logical units like Amazon's EC2 Compute Unit.

In order to devise a VM placement algorithm that can effectively cope with the complexity of multicore scheduling, new algorithmic techniques are necessary. Most previous research used either fast greedy heuristics with no guarantees about the quality of the solutions they deliver, or exact methods that do not scale to practical problem sizes. In this paper, we try to find some middle ground. We argue that in a typical scenario, it is acceptable to spend 1-2 minutes on VM placement optimization. Moreover, there is typically a lot of available computational capacity in a DC that can be used for running the optimization algorithm. Hence, supposing that the optimization algorithm can be sufficiently parallelized, a significant portion of the search space can be investigated.

We identified *constraint programming* as an ideal framework that allows us to (i) formulate complex constraints like multicore scheduling in a natural way; (ii) makes it possible to define VM placement as a global optimization problem with a well-defined objective function; and (iii) by incorporating different heuristics, enables a balance between solution quality and running time. Parallelization is handled by splitting the search space into as many parts as the number of parallel resources available for running the algorithm, and then taking the best of the solutions found in each part.

The contributions of the paper are:

- Identification of the aspects that need to be taken into account for the placement of multicore VMs on multicore PMs.
- Definition of a problem formulation of VM placement, in which both PMs and VMs can have multiple cores. Besides, the problem formulation includes the cost of migration of VMs as well as the cost of SLA violations.
- Combination of global search with heuristics in a constraint programming framework, in order to balance solution quality and solving time.
- A simulation-based empirical study to show that the proposed algorithms deliver significantly better results compared to a typical non-multicore-aware heuristic proposed previously in the literature.

The rest of the paper is organized as follows. Section 2 reviews previous work, followed by a discussion on the issues faced by multicore VM placement in Section 3. Sections 4 and 5 describe the used problem model and our algorithms, respectively. An empirical evaluation is presented in Section 6, and Section 7 concludes the paper.

2 Previous work

In recent years, the VM placement problem has received much attention [31]. Typical problem formulations almost always include computational capacity of PMs and computational load of VMs as a single dimension. In fact, in many works, this is the only dimension that is considered [3], [4], [5], [6], [7], [8], [9], [12], [20], [25], [28], [44], [45]. Other authors included, beside the CPU,

also some other resources like memory, I/O, storage, or network bandwidth [11], [18], [19], [42], [48].

Multicore processors were hardly taken into account. Some authors suggested to use the number of cores as a metric, i.e., the number of available cores of a PM and the number of cores that VMs occupy [26], [34], [41]. However, this approach does not support the sharing of a pCPU by multiple vCPUs.

Several different objective or cost functions have been proposed. The number of active PMs is often considered because it largely determines the total energy consumption [5], [9], [12], [19], [34], [45], [48]. SLA violations may lead to penalties that also need to be minimized [4], [9], [19], [20], [45], [48]. Usually, it is assumed that an SLA violation happens if a PM is overloaded and thus the impacted VMs are not assigned the amount of resources that they would need [6], [9], [12], [42], [45], [47]. Another important factor that some works considered is the cost of migration of VMs [6], [12], [19], [41], [44].

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 [4], [5], [9], [20], [26], [29], [44], [45], other greedy heuristics [35], [47] and straight-forward selection policies [3], [37], as well as meta-heuristics [18], [19].

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 [3], binary integer programming [11], [29], mixed integer non-linear programming [20], and pseudo-Boolean optimization [34]. Unfortunately, all these methods suffer from a scalability problem, limiting their applicability to small-scale problem instances.

Optimal application placement on multicore architectures has been considered for MPI processes [24]. However, that work was not in the context of virtualization and VM placement; moreover, the sharing of a processor core my multiple processes was not considered. Ahn et al. considered VM placement in a multicore context from the point of view of minimizing contention of resources shared by co-located VMs like caches, and showed that there is significant performance difference between the best and worst placement especially for memory-intensive workloads [1]. However, that work did not take into account several other important characteristics of the VM placement problem, like the capacity of the PMs or SLA violations.

3 Multicore scheduling issues

Most of the existing VM placement algorithms only consider the total CPU capacity of PMs and the total CPU load of VMs, and assume that a set of VMs can be mapped onto a PM if and only if their total CPU load is not greater than the CPU capacity of the PM. In reality, the allocation of vCPUs on pCPUs is a complex task to be solved by the scheduler of the hypervisor. This will be referred to as *core scheduling*. Beyond the placement of the VMs, the results of core scheduling also impact overall performance and costs. Since core scheduling follows only after VM placement, core scheduling is constrained by the allocation of VMs to PMs, and it may lead to results not anticipated by a non-multicore-aware VM placement algorithm. Even if both VM placement and core scheduling work optimally, if the VM placement does not take into account the schedulability of vCPUs on pCPUs, overall results may be sub-optimal.

In the following, we review some scheduling issues that can adversely impact performance and/or costs if not taken into account by the VM placement algorithm.

3.1 High sequential compute demand

Suppose a VM hosts a single-threaded application and requires a single vCPU with 2000 MIPS to perform its tasks in due time. A VM placement algorithm that only looks at the total CPU capacity of PMs may decide to place this VM on a PM with two pCPUs, offering 1000 MIPS per pCPU. This seems to be a good decision for the VM placement algorithm because the total CPU capacity of the PM is 2000 MIPS, just enough for the given VM. However, since the application is single-threaded, the VM will not be able to take advantage of the two available pCPUs. It can only use one pCPU, and will thus receive only 1000 MIPS capacity, leading to a performance degradation of factor 2.

3.2 vCPU migration vs. pinning

The hypervisor may decide to dynamically re-arrange the mapping of vCPUs to pCPUs. At any given time, each vCPU is allocated to just one pCPU, but through regular core migrations, a vCPU can be served by multiple pCPUs when regarded over a given period of time.

Core migration has an overhead, consisting not only of the time needed to transfer the state of the vCPU from one pCPU to the other, but also related to cache locality. The latter is important if each pCPU has its own L1 (and possibly also L2) cache, or a cache slice in a common cache that it can access faster [38]. Migrating the vCPU has the consequence that data has to be reloaded into the cache of the new core. For these reasons, in some situations it is more beneficial to pin vCPUs to specific pCPUs, thus avoiding core migrations [24].

The exact impact of vCPU migration on VM performance depends on several factors. Kim et al. showed that, for the case of an under-committed CPU, the default vCPU relocation behavior of the Xen scheduler results in up to 15% performance overhead – but it is beneficial for the over-committed case [27]. For asymmetric processors, DeVuyst et al. reported core migration overhead of up to 44 msec [16]. For some applications, the overhead of core migration may be negligible, whereas for others it can be a major problem.

Another reason for pinning a vCPU to a specific pCPU may be per-pCPU software licensing, see e.g. [46]. In this case, the vCPUs running the given application should be confined to a number of pCPUs for which the appropriate license is available.

3.3 Further issues

Further multicore scheduling issues that may be relevant for VM placement include:

- Multi-socket and NUMA architectures
- Hyper-threading
- Dedicated cores
- Asymmetric processors

Details on these issues are presented in the supplemental material available online.

4 A POSSIBLE PROBLEM FORMULATION

In Section 3, a number of scheduling issues were mentioned that may have an impact on the cost and performance of VM placement. What is common in these issues is that a VM placement algorithm that only considers the total CPU capacity of PMs and the total CPU load of VMs is bound to make bad decisions. Therefore, the remainder of this paper focuses on *enhancing VM placement by incorporating awareness of core scheduling*. This is possible if the mapping of VMs on PMs is combined with mapping of vCPUs on pCPUs, resulting in an optimization problem that combines VM placement and – a somewhat simplified version of – core scheduling.

For sure, adding more details to the VM placement problem will considerably enlarge the search space of this already difficult optimization problem. Therefore, scalability is a major concern and will be investigated later on in more detail.

Which of the issues mentioned in Section 3 are relevant in a given practical situation depends on several factors, including workload characteristics (e.g., CPU-bound vs. memory-bound applications) and the features of the infrastructure (e.g., UMA vs. NUMA architecture). Each possible combination of the issues of Section 3 may be formulated as a slightly different optimization problem (and that list is not intended to be exhaustive). In the following, one possible such problem formulation will be used, which addresses some of those issues. But more importantly, our aim is to demonstrate that *it is possible to combine VM placement with core scheduling*. This problem model – and the resulting algorithms that will be presented afterwards – can be modified as necessary to accommodate different scheduling issues.

4.1 Formal problem model

We are given a set P of PMs. For a PM $p \in P$, the set of its cores (pCPUs) is denoted by $PC(p)$; the computational capacity of each core of p is $pcc(p) \in \mathbb{R}^+$. Besides, we are given a set V of VMs. For each VM $v \in V$, the set of its cores (vCPUs) is denoted by $VC(v)$. $PC := \bigcup \{PC(p) : p \in P\}$ denotes the set of all pCPUs, and similarly, $VC := \bigcup \{VC(v) : v \in V\}$ denotes the set of all vCPUs.

We assume that the VM-to-PM mapping is regularly re-optimized based on changes in the VMs' load and potentially in PMs' availability. There is a current mapping of VMs to PMs, represented by a function $map_0 : V \to P$. Furthermore, we assume there is an estimate of the VMs' load for the next period: for each vCPU $vc \in VC$, $vcl(vc) \in \mathbb{R}^+$ denotes its estimated load. Based on the new estimates, our goal is to compute a new mapping $map : V \to P$.

The cost of a mapping is comprised of three components: the number of active PMs, the number of migrations, and the number of overloaded pCPUs. A PM is called active if at least one VM is mapped to it. PMs that are not active can be switched to sleep mode. Therefore, to save energy, we should aim at minimizing the *number of active PMs*, denoted by $A(map)$.

A VM $v \in V$ must be migrated if $map(v) \neq map_0(v)$. Since migration incurs significant additional load for the involved machines as well as for the network, it is important to also minimize the *number of migrations*, denoted by $M(map)$.

If a pCPU is overloaded, then the vCPUs it serves do not obtain the required computational capacity, which may lead to performance degradation and to an SLA violation. Thus, it is important to keep the number of pCPU overloads low. In order to determine the number of overloaded pCPUs, we need a mapping of vCPUs to pCPUs, denoted by

$$cmap : VC \to PC. \tag{1}$$

The core mapping problem is described by the following rules:

1) Each vCPU of each VM must be mapped on exactly one of the pCPUs of the PM that accommodates the VM. Formally:
$$\forall v \in V,\ \forall vc \in VC(v) : cmap(vc) \in PC(map(v)). \tag{2}$$
2) A pCPU can accommodate multiple vCPUs, even belonging to multiple VMs.
3) The vCPUs of a VM can be served by the same or by different pCPUs of the PM.
4) A vCPU cannot be split on multiple pCPUs.

For a pCPU $pc \in PC$, $cmap^{-1}(pc)$ is the set of all vCPUs mapped on this pCPU. Hence, pCPU $pc \in PC$ of PM p is overloaded if and only if

$$\sum_{vc \in cmap^{-1}(pc)} vcl(vc) > pcc(p). \tag{3}$$

The *number of overloaded pCPUs* is denoted by $S(cmap)$.

The cost function that we must minimize is given by

$$F(map, cmap) = \alpha \cdot A(map) + \mu \cdot M(map) + \sigma \cdot S(cmap), \tag{4}$$

where α, μ, and σ are non-negative constants determining the relative weight of the three sub-goals. Our aim is to find mappings map and $cmap$ that minimize F.

TABLE 1
Summary of notation

Notation	Explanation
P	set of PMs
$PC(p)$	set of pCPUs of PM p
PC	set of all pCPUs of all PMs
$pcc(p)$	computational capacity of each core of PM p
V	set of VMs
$VC(v)$	set of vCPUs of VM v
VC	set of all vCPUs of all VMs
$vcl(vc)$	estimated load of vCPU vc for the next period
map_0	current mapping of VMs to PMs
map	new mapping of VMs to PMs (to be determined)
$cmap$	mapping of vCPUs to pCPUs (to be determined)
$A(map)$	number of active PMs
$M(map)$	number of migrations
$S(cmap)$	number of overloaded pCPUs
$F(map, cmap)$	cost function to be minimized
α	weight of the number of active PMs
μ	weight of the number of migrations
σ	weight of the number of overloaded pCPUs

Table 1 gives a summary of the used notations.

4.2 Discussion and variations

As mentioned earlier, the suggested problem model is just one possibility. Its main feature is that it combines VM placement (determining the map function) with a simplified version of core scheduling (determining the $cmap$ function). The main simplification is that time-dependent dynamic features of scheduling are not included. Nevertheless, we believe this model is a good compromise: introducing the $cmap$ function without time-dependence already allows us to reason about specific cores and thus address the issues mentioned in Section 3. Also including time-dependence could be a next step, but it would again considerably blow up the search space with limited improvement in precision.

4.2.1 Core mapping rules

In Section 4.1, four rules were introduced to define what is allowed and what is not allowed in core scheduling. It can be seen easily that Rules 1)-4) state exactly the same as equations (1)-(2).

Rule 3 may be seen as too permissive as it allows multiple vCPUs of the same VM to be accommodated by the same pCPU, which might be undesirable for applications that need true parallelism. In this case, the model can be extended with an additional constraint to exclude this: $\forall v \in V,\ \forall vc_i \neq vc_j \in VC(v) : cmap(vc_i) \neq cmap(vc_j)$.

Rule 4 is the result of the reasoning in Section 3.2 about the uses of pinning. It is also possible that this reasoning is valid for some critical VMs but not for all. This would require a somewhat more complicated problem formulation, in which the above model of core scheduling is used for the critical VMs, but for the others, also a fractional mapping is allowed (e.g., a vCPU can be mapped to the extent of 70% to one pCPU and to 30% to another one). It is not very difficult to extend the above problem formulation in this way, but it makes the formalization more cumbersome.

4.2.2 Cost function

Minimizing the number of active PMs, the number of migrations, and the number of SLA violations due to resource overloads are all typical objectives that have been widely used in VM placement research. (But an important difference is that we consider the overload of individual pCPUs, whereas previous work considered overload of the CPU as a whole.) For example, the work of Beloglazov and Buyya uses these metrics [6], whereas other works use a subset of these metrics [9], [20], [44].

There are many variations concerning the details of these metrics. For example, instead of just the number of active PMs, one could also consider their total energy consumption (taking into account the different power efficiency of the PMs as well as load-dependent dynamic consumption) because the real objective is energy minimization, and minimizing the number of active PMs is just a way to achieve that. Similarly, instead of the number of migrations, one could consider the total cost of migrations, where the cost of a migration may depend on factors such as the memory image size of the given VM. All these concerns are orthogonal to our work; our problem formulation could easily be modified to take them into account if necessary.

A further question is how to define a proper optimization problem based on multiple cost metrics. One possibility is to formulate a multi-objective optimization problem and look for Pareto-optimal solutions [18], [43]. Another approach is to constrain all but one of the metrics and optimize according to the remaining one [2], [11]. The third possibility is to combine multiple metrics into a single objective function, for example as a weighted sum of the metrics [21], [25], [37]. We chose this third approach but the other two would also be possible.

4.2.3 Further issues

Further discussion on the appropriateness and possible extensions of the problem model is provided in the supplemental material.

5 Algorithms

We devise multiple algorithms for the problem defined in Section 4.

5.1 The case for constraint programming

Constraint programming (CP) has already been proposed for related problems [17], [22], [23] and shown to be a useful tool for deriving high-quality solutions in acceptable time. However, CP does not belong to the really popular approaches used for VM placement: algorithms in this field almost exclusively fall into one of three categories: greedy heuristics, proprietary heuristics, and (mixed) integer programming.

Our problem model includes a strong scheduling component: a highly combinatorial problem for which CP is known to be an excellent approach [10]. Besides, CP has the advantage that variations such as the ones discussed in Section 4.2 can be incorporated with relative ease by posting the appropriate constraints.

5.2 A very short introduction to CP

Since CP is not so widely known, we give a brief introduction to the most fundamental concepts. For further details, we refer to the available rich literature, e.g. [10] and references therein. As implementation framework, we used the CLPFD (constraint logic programming over finite domains) library [15] of SICStus Prolog 4.2.3.

A typical constraint program consists of the following main steps:

1) Definition of the *variables* and their *domains*. The domain of a variable is the set of possible values that the given variable can be assigned.
2) Posting the *constraints*. Each constraint contains one or more variables and defines a relation that those variables must fulfill.
3) *Search*. This is usually the most time-consuming phase in which the CP engine searches the space of possible variable assignments to find a solution that fulfills all constraints or, in the case of an optimization problem, a solution fulfilling all constraints and maximizing or minimizing a given objective function. The search procedure is a backtrack search algorithm that can be customized in several ways to achieve good performance.

A key concept is *pruning*. When a variable x is assigned a value, the constraints involving x wake up and propagate the consequences of this assignment. As a result, some values of another variable y may become infeasible; these are then pruned from the domain of y. This change to the domain of y may in turn wake up further constraints that may prune further values from the domain of a third variable and so on. Pruning infeasible values as early as possible helps to keep the size of the search tree manageable, thus increasing efficiency.

CLPFD supports a wide range of constraints, including arithmetic, propositional, and combinatorial constraints. It also supports *reification* with which the truth value of a given constraint can be mirrored in a Boolean variable. For example, the constraint `(X #> Y) #<=> B` expresses that `B` must have the value 1 if `X` is greater than `Y` and 0 otherwise. Boolean variables are normal variables with the domain `{0,1}` and can be manipulated just as any other integer variables.

5.3 Pure CP solution

Our first approach consists of formulating the problem using CP (see Fig. 2).

Two sets of primary variables are used. VMs are numbered from 1 to $n = |V|$ and PMs are numbered

1: Define variables and their domains
2: $1 \le j \le n:\ x_j \in \{1, \dots, m\}$
3: $1 \le jc \le nc:\ y_{jc} \in \{1, \dots, mc\}$
4: $1 \le j \le n:\ z_j \in \{0, 1\}$
5: $1 \le jc \le nc, 1 \le ic \le mc:\ u_{jc,ic} \in \{0, 1\}$
6: $1 \le ic \le mc:\ w_{ic} \in \{0, 1\}$
7: $A, M, S, F \in \mathbb{N}$
8: End
9: Define constraints
10: x_j and y_{jc} are compatible (using `table`)
11: $A = |\{x_j : 1 \le j \le n\}|$ (using `nvalue`)
12: $x_j \neq map_0(vm_j) \Leftrightarrow z_j$
13: $M = \sum_{j=1}^{n} z_j$ (using `sum`)
14: $y_{jc} = ic \Leftrightarrow u_{jc,ic}$
15: $\sum_{jc=1}^{nc} u_{jc,ic} \cdot vcl(vc_{jc}) > pcc(pc_{ic}) \Leftrightarrow w_{ic}$ (using `scalar_product`)
16: $S = \sum_{ic=1}^{mc} w_{ic}$ (using `sum`)
17: $F = \alpha \cdot A + \mu \cdot M + \sigma \cdot S$
18: End
19: Perform search
20: k parallel threads
21: each running a randomized backtrack search
22: with timeout τ
23: minimizing F
24: End
25: Return solution with lowest F

Fig. 2. Pseudo-code of the pure CP algorithm

from 1 to $m = |P|$. For $1 \le j \le n$, variable x_j encodes the PM that VM j should be mapped to; the domain of each x_j is $1, \dots, m$. vCPUs are numbered consecutively from 1 to $nc = |VC|$ and pCPUs consecutively from 1 to $mc = |PC|$. Then, for each $1 \le jc \le nc$, the variable y_{jc} encodes the pCPU that should accommodate vCPU jc; the domain of each y_{jc} is $1, \dots, mc$.

The x_j variables encode the *map*, the y_{jc} variables the *cmap* function. It must be ensured that the two mappings are consistent, meaning that the vCPUs of a VM can be mapped by *cmap* only on the pCPUs of the PM where the VM is mapped by *map* (Rule 1 in Section 4.1). This can be elegantly and efficiently assured by means of a single `table/2` constraint, one of the built-in combinatorial constraints of CLPFD [14]. In order to define a k-ary finite relation, `table(A,B)` can be used, where both `A` and `B` are lists, and each element of both `A` and `B` is a list of length k. The relation itself is defined by `B`: the elements of the list `B` are the valid k-tuples that form the relation. For compactness, the lists within `B` may also contain ranges. For example, `B=[[1,10],[2,20..22]]` means that the valid tuples are (1,10), (2,20), (2,21), (2,22). The elements of `A` are tuples of variables; the constraint ensures that each of these tuples of variables will satisfy the relation specified by `B`. In our case, `table/2` is used to define the binary relation between each y_{jc} variable and the corresponding x_j variable. Therefore, `A` will be a list of all pairs (x_j,y_{jc}), where the vCPU jc belongs to VM j. The elements of `B` are of the form [*i*,*ic_min*..*ic_max*], where $1 \le i \le m$ specifies a PM, and *ic_min* and *ic_max* denote the index of its first and last pCPU, respectively, in the

consecutive numbering of all pCPUs in PC. This way, `table(A,B)` ensures indeed that the mapping of VMs and the mapping of vCPUs will be compatible.

From the x_j variables, the number of active PMs can be calculated easily: this is the number of different values taken by the x_j variables. For this purpose, the built-in `nvalue/2` constraint can be used. In general, the constraint `nvalue(Num,Vars)`, where `Num` is either a variable or an integer and `Vars` is a list of variables, assures that the number of different values taken by `Vars` equals `Num`. In our case, this can be used with the variable representing the number of active PMs as `Num` and the list of the x_j variables as `Vars`.

For calculating the number of migrations, we need to define a set of secondary variables: for each $1 \leq j \leq n$, z_j is a Boolean variable that has the value 1 if and only if VM j is migrated, i.e. $x_j \neq map_0(vm_j)$. The z_j variables are determined from the x_j variables using reification, and the number of migrations is calculated as the sum of the z_j variables using the built-in `sum/3` constraint.

For determining the number of overloaded pCPUs, our first implementation follows a similar logic. We define further secondary variables: for each $1 \leq jc \leq nc$, $1 \leq ic \leq mc$, the Boolean variable $u_{jc,ic}$ encodes whether vCPU jc is mapped on pCPU ic. The $u_{jc,ic}$ variables can be determined from the y_{jc} variables using reification, and based on the $u_{jc,ic}$ variables, the total load of a pCPU can be calculated using the `scalar_product/4` built-in constraint. Based on the loads of the pCPUs, the number of pCPU overloads can be determined using another round of reification and summation; the auxiliary variables w_{ic} ($i \leq ic \leq mc$) encode whether pCPU ic is overloaded.

Given the three cost factors A, M, and S, the cost function F can be calculated and the constraint engine instructed to find a solution that minimizes this cost function, using the `minimize` option of the `labeling/2` built-in procedure. Only the primary variables are labeled, since the value of the secondary variables can be inferred from them.

We customized the search procedure of the constraint engine, which uses an exhaustive backtrack search by default, to make it more efficient. For selecting the next variable to branch on, we use the first-fail heuristic: the variable with the smallest domain is selected. For enumerating the possible values of the chosen variable, we implemented a custom randomized procedure, so that different runs of the search will likely explore different parts of the search space. This is a great opportunity for parallelization: we run k parallel searches, each with timeout τ, where k depends on the number of parallel processing units that are available and τ is a pre-defined amount of time that we are willing to wait for the result. At the end, we return the best of the results found by the k searches.

5.4 Enhanced pure CP solution

Preliminary experiments with our first CP implementation revealed a bottleneck that caused scalability issues. As mentioned above, for calculating the number of overloaded pCPUs, $nc \cdot mc$ auxiliary variables (the $u_{jc,ic}$ variables) had to be introduced and calculated with a similar number of constraints. It should be noted that otherwise our CP model scales linearly with input size, only this part scales quadratically. As the number of pCPUs and vCPUs increases, this becomes a problem in terms of both memory consumption and the time required for manipulating the high number of constraints.

Therefore, in a second version of our CP solution, we implemented a dedicated global constraint for calculating the number of pCPU overloads directly from the y_{jc} variables, using the possibilities offered by CLPFD for defining user-level global constraints. This way, there is no need for the quadratic number of auxiliary variables and so the CP model scales linearly with input size.

Compared to the pseudo-code in Fig. 2, lines 5-6 and 14-16 are disappear. Instead of lines 14-16, S is calculated directly from the y_{jc} variables using the dedicated constraint `nr_overloads`.

5.5 Greedy VM-to-PM mapper

As a baseline, we also investigate a greedy algorithm inspired by well-known bin-packing heuristics. Specifically, we use the algorithm of Beloglazov et al. that was shown to be quite effective for the VM-to-PM mapping problem in practice and can be seen as a typical example of a state-of-the-art non-multicore-aware VM placement algorithm [4], [5]. It aims at minimizing power consumption by minimizing the number of active PMs as well as minimizing the number of migrations while obeying the (one-dimensional) capacity constraints of the PMs. The algorithm works by first removing the VMs from lightly used PMs so that they can be switched off and removing a minimal set of VMs from overloaded PMs so that they will not be overloaded (a so-called minimal relieving set [32]). In a second phase, the algorithm finds a new accommodating PM for the removed VMs using the Modified Best Fit Decreasing (MBFD) heuristic[2]. See Fig. 3 for more details.

The original algorithm does not define a mapping of vCPUs on pCPUs. Since we need that mapping for evaluating the number of pCPU overloads, we extended the algorithm with a further step, in which each vCPU is mapped on a randomly selected pCPU of the accommodating PM (lines 16-19).

2. In its original form, the algorithm chooses for each VM the PM that has sufficient capacity and that would lead to the lowest increase in power consumption; in case of a tie, it chooses the first such PM. Since in our model only the number of active PMs counts, not their exact power consumption values, the algorithm simply chooses the first PM with sufficient capacity.

```
1: L ← ∅
2: for all p ∈ P do
3:     if p is underloaded then
4:         remove all VMs from p
5:         add the VMs to L
6:     else if p is overloaded then
7:         remove a minimal relieving set of VMs from p
8:         add the VMs to L
9:     end if
10: end for
11: sort L in descending order of CPU load
12: for all v ∈ L do
13:     for all p ∈ P do
14:         if load(p) + load(v) ≤ cap(p) then
15:             allocate v on p
16:             for all vc ∈ VC(v) do
17:                 choose pc ∈ PC(p) randomly
18:                 allocate vc on pc
19:             end for
20:             break
21:         end if
22:     end for
23: end for
```

Fig. 3. Pseudo-code of the greedy algorithm

5.6 Hybrid approaches

Our initial experiments reinforced our expectation that the greedy algorithm is much faster than the CP approach, but, since it does not account for core mapping, it leads to a high number of pCPU overloads. In the following, we devise possible hybrid algorithms (which will later be denoted as hybrid1, hybrid2, and hybrid3) to combine the strengths of the two approaches.

5.6.1 Greedy algorithm with schedulability analysis

This is essentially the same as the above greedy algorithm, with a single modification. When looking for a new accommodating PM for a given VM, the original MBFD heuristic determines whether the VM fits on a PM by simply checking whether the PM's current load plus the VM's load is not greater than the PM's capacity (line 14 in Fig. 3). The modified algorithm performs instead a schedulability analysis, i.e., it verifies that the vCPUs of the VMs that are currently mapped to this PM, together with the vCPUs of the new VM, can be mapped to the pCPUs of the PM. This analysis is carried out using constraint programming. All other steps of Fig. 3 remain unchanged; line 14 is replaced by a call to the schedulability check shown in Fig. 4.

5.6.2 Greedy algorithm with optimized core mapping

This is the same as the normal greedy algorithm, but after the VM-to-PM mapping has been established, the mapping of cores is not done randomly, but by attempting to find an optimal mapping of cores for each PM and its accommodated VMs. The core mapping is found using constraint programming, with the objective of minimizing the number of pCPU overloads. Compared to Fig. 3, lines 16-19 are substituted by a call to the core mapping procedure depicted in Fig. 5.

```
1: procedure SCHEDULABLE(PM p, set of VMs W)
2:     Let VC(W) = ⋃{VC(v) : v ∈ W}
3:     Number vCPUs in W from 1 to |VC(W)|
4:     Number pCPUs of p from 1 to |PC(p)|
5:     Define variables and their domains
6:         1 ≤ jc ≤ |VC(W)| : y_jc ∈ {1, ..., |PC(p)|}
7:     End
8:     Define constraints
9:         no core is overloaded
10:    End
11:    Perform search
12:        standard backtrack search algorithm
13:    End
14:    return whether solution has been found
15: end procedure
```

Fig. 4. Pseudo-code of the schedulability analysis

```
1: procedure COREMAPPING(PM p, set of VMs W)
2:     Let VC(W) = ⋃{VC(v) : v ∈ W}
3:     Number vCPUs in W from 1 to |VC(W)|
4:     Number pCPUs of p from 1 to |PC(p)|
5:     Define variables and their domains
6:         1 ≤ jc ≤ |VC(W)| : y_jc ∈ {1, ..., |PC(p)|}
7:         S ∈ ℕ
8:     End
9:     Define constraints
10:        S equals the number of core overloads
11:    End
12:    Perform search
13:        standard backtrack search algorithm
14:        minimizing S
15:    End
16:    return best solution found
17: end procedure
```

Fig. 5. Pseudo-code of the optimized core mapping procedure

5.6.3 Greedy algorithm with schedulability analysis and optimized core mapping

This is a combination of the above two possibilities: the MBFD algorithm is extended with schedulability analysis to account for multicore scheduling in its decisions, and afterwards, the mapping of cores is determined using the constraint programming approach, explicitly minimizing the number of pCPU overloads. Compared to Fig. 3, line 14 is replaced by a call to the schedulability check of Fig. 4, while lines 16-19 are replaced by a call to the core mapping routine of Fig. 5.

5.7 Two-stage CP algorithm

This is a relaxation of the pure CP approach using ideas from the extended greedy algorithm. In the first stage, we search the space of possible VM-to-PM mappings with the aim of minimizing the simplified cost function $F'(map) = \alpha \cdot A(map) + \mu \cdot M(map) + \sigma \cdot S'(map)$. At this stage, core mapping is not considered yet. The number of overloads (S') is calculated at the level of PMs: a PM is considered overloaded if the total load of all vCPUs of all VMs mapped to the PM exceeds the PM's total computing capacity (similar to the one-dimensional checks of the original MBFD algorithm).

In the second stage, when the VM-to-PM mapping is already determined, the optimal mapping of cores (wrt. the number of pCPU overloads) is found for each PM and the VMs it accommodates.

Both stages use CP (see Fig. 6), aiming to find the global optimum for the given cost function, but with a timeout. Because of the split of stages, even if both stages find their optima, this is not necessarily the optimum for the whole problem. However, the rationale behind the split of stages is that this way, the search space is dramatically reduced, thus allowing a more effective search.

1: Define variables and their domains
2: $1 \le j \le n:\ x_j \in \{1,\dots,m\}$
3: $1 \le j \le n:\ z_j \in \{0,1\}$
4: $A, M, S', F' \in \mathbb{N}$
5: End
6: Define constraints
7: $A = |\{x_j : 1 \le j \le n\}|$
8: $x_j \neq map_0(vm_j) \Leftrightarrow z_j$
9: $M = \sum_{j=1}^{n} z_j$
10: S' equals the number of overloaded PMs
11: $F' = \alpha \cdot A + \mu \cdot M + \sigma \cdot S'$
12: End
13: Perform search
14: standard backtrack search algorithm
15: minimizing F'
16: End
17: /* *Second stage* */
18: **for all** $p \in P$ **do**
19: call CoreMapping for p and its hosted VMs
20: **end for**

Fig. 6. Pseudo-code of the twostage algorithm

5.8 Limiting the runtime

As mentioned earlier, the search procedure of the pure CP algorithm is tailored so that it makes k independent searches, each with time limit τ. This way, if k parallel processing units are available for executing the algorithm, it can finish in τ time. For the sake of comparability, the runtime of the other algorithms needs to be limited similarly.

The greedy algorithm is very fast, so that no time limit is needed. In the hybrid1 algorithm, the schedulability analysis is carried out a high number of times and may be relatively time-consuming; therefore, its running time has to be limited. Let ν denote the number of VMs to migrate and m the number of PMs. For each VM to migrate, the possible PMs are tried – using schedulability analysis – until one is found where the VM fits. Assuming that on average $m/2$ PMs must be tried, altogether $\nu \cdot m/2$ runs of the schedulability analysis procedure are necessary. Assuming again k parallel processing units and an overall time limit of τ, the time limit per run is $2 \cdot k \cdot \tau/(\nu \cdot m)$.

Similarly, in the hybrid2 algorithm, the running time of the search procedure for optimized core mapping needs to be limited. Since this procedure is run for each of the m PMs once, and can be again parallelized, the resulting time limit per run is $k \cdot \tau/m$. In the hybrid3 algorithm, the running time of both the schedulability analysis searches and the core mapping searches must be limited; the corresponding limits can be calculated similarly as above. Finally, in the twostage algorithm, the time budget of τ must be split between the two stages; in our current implementation, this is achieved by simply halving it. Moreover, the $\tau/2$ time available for the second stage must be split between m independent core mapping searches, resulting in time limits of $k \cdot \tau/(2 \cdot m)$ for each search.

6 Simulation results

In order to foster the reproducibility of the results, the used programs as well as measurement data are publicly available from http://www.cs.bme.hu/~mann/data/multicore_VM_placement/.

6.1 Experimental setup

All measurements were carried out on a notebook computer with Intel i3-3110M CPU running at 2.40 GHz and 4GB RAM, with Windows 7 Enterprise.

Unless otherwise noted, we used $\tau = 60sec$ and $k = 10$, i.e., the algorithms would finish within 1 minute, assuming 10 parallel processing units for running them.

The cost of a solution is evaluated using the cost function defined in Section 4.1, with the following weights: $\alpha = 3$, $\mu = 1$, $\sigma = 2$.

TABLE 2
PM types

Type	Number of cores	Capacity per core
1	2	1000
2	4	2000
3	8	3000

In the first experiments, synthetic data were used. We model a DC with three types of PMs having different capacity, as shown in Table 2. Each PM in the DC belongs to one of the three types, with each type having probability 1/3. VMs are randomly generated to have 1, 2, or 4 cores (each with equal probability), and the load of each vCPU is generated as a uniform random number between 100 and 1500. For the initial VM-to-PM mapping, each VM is mapped to a PM selected in a uniform random manner.

6.2 Experiment 1: density

In a first experiment, we fix the number of PMs to 200, and vary the number of VMs from 50 (lightly loaded DC) to 1200 (heavily loaded DC) in steps of 50. For each instance, all the six algorithms presented in Section 5 are run. The results are shown in Fig. 7.

As can be seen, the greedy and hybrid1 algorithms perform very similarly to each other, and not too well

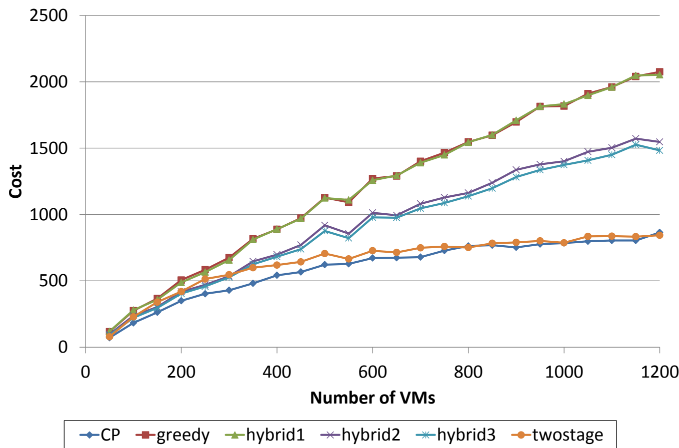


Fig. 7. Cost of the solution delivered by the different algorithms for 200 PMs and varying number of VMs

compared to the other algorithms. This can be explained by the fact that these two algorithms do not strive for an optimal mapping of vCPUs on pCPUs. The hybrid2 algorithm outperforms the ones mentioned previously, showing the importance of a powerful search method for finding an appropriate mapping of vCPUs on pCPUs. The hybrid3 algorithm, which adds schedulability analysis to hybrid2, consistently performs even better. This is interesting because the schedulability analysis did not help much in the case of hybrid1 over greedy: apparently, a-priori schedulability analysis is only useful in combination with intelligent a-posteriori scheduling.

The CP algorithm performs very well: in almost all cases, it yields the best results among all investigated algorithms. On one hand, this is not surprising, because it performs a systematic search and explicitly minimizes the given objective function. On the other hand, the excellent performance of the CP algorithm could not be taken for granted since the applied time limit allows it to only scan a tiny fraction of the vast search space.[3] The empirical results show that the CP algorithm can quickly achieve very good results, without the need for an exhaustive search.

Finally, the twostage algorithm performs also quite well, especially for VM numbers greater than 300, where it delivers results that are almost as good as the ones of the CP algorithm. This shows that constraint programming, together with appropriate heuristic splitting of the search space, can be indeed very powerful. For low densities, the hybrid3 algorithm is also quite good (better than twostage), but after about 300 VMs, where the problem starts to be highly constrained, the twostage algorithm is clearly better. For very high densities, the results delivered by the CP and twostage algorithms are almost 60% better than those of the greedy algorithm, underlining the importance of systematic search for highly constrained problems.

TABLE 3
Detailed results for 200 PMs and 600 VMs

Algorithm	Active PMs	Migrations	pCPU overloads	Total cost
CP	178	138	0	672
greedy	148	355	236	1271
hybrid1	149	357	227	1258
hybrid2	148	355	107	1013
hybrid3	149	357	87	978
twostage	191	6	74	727

The costs in Fig. 7 are with respect to the cost function defined in Section 4.1. It is also interesting to look at the individual components of the cost function. As an example, Table 3 shows the details for 200 PMs and 600 VMs. As can be seen, the greedy algorithm is characterized by too aggressive consolidation: it results in a low number of active PMs, but many pCPU overloads. The latter shortcoming is effectively mitigated by the hybrid2 and especially the hybrid3 algorithms: they lead to a reduction of 55% respectively 63% in the number of pCPU overloads, virtually without affecting the other two cost components. The twostage algorithm also leads to a low number of pCPU overloads; the higher number of active PMs is compensated by a significantly reduced number of migrations. Finally, the CP algorithm which explicitly minimizes the number of overloads in its systematic global search, achieves a very low number of overloads (0 in this case). Concerning the number of migrations, it is less effective than the twostage algorithm, but still much better than the other competing algorithms, leading to the overall best solution.

6.3 Experiment 2: scalability

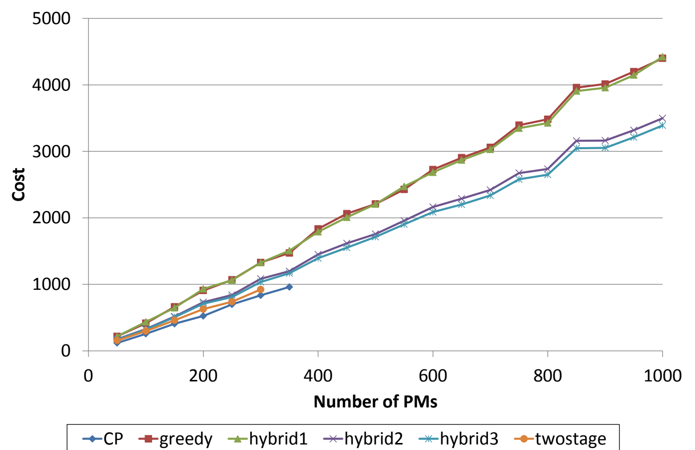


Fig. 8. Cost of the solution delivered by the different algorithms for varying input sizes, where the number of VMs is twice the number of PMs

The next experiment aims at evaluating the scalability of the algorithms. For this purpose, we fixed the n/m

3. As an example: for 200 PMs and 600 VMs, there are 200^{600} possible VM-to-PM mappings: an enormous number, and core scheduling makes the search space even much larger.

ratio[4] to 2, and varied m from 50 to 1000, in steps of 50, see Fig. 8. Although in the beginning, the CP and twostage algorithms deliver the best results, they do not scale well. Their running time is bounded, and since they perform systematic search, it can happen that they do not find any solution within the given time limit. For CP, this is the case for inputs with $m > 350$, for twostage, this happens for $m > 300$. The other algorithms scaled well even to the biggest investigated inputs ($m = 1000$, $n = 2000$). The greedy and hybrid1 algorithms are consistently outperformed by the remaining algorithms, with hybrid3 delivering consistently the best results, which are roughly 25% better than those of the greedy algorithm.

Of course, there are also much bigger DCs, with tens or hundreds of thousands of PMs. It is clear that our algorithms featuring a very detailed model of PMs and VMs cannot be applied in that scale. From the above experiments, we can see that the proposed algorithms work well for some hundreds of PMs (e.g., CP works for up to 300 PMs, hybrid3 works for up to 1000 PMs). Thus, they are applicable to DCs of small and medium-sized organizations which are actually responsible for the major part of carbon emission caused by DCs [33]. In big DCs, hierarchical VM placement algorithms may be used, where on the higher hierarchy levels only aggregate information is used. Our algorithms utilizing more detailed information can be used on the level of racks or clusters; fortunately, they do scale to the size necessary for that.

6.4 Experiment 3: runtime vs. quality

In order to guarantee acceptable runtimes, the presented algorithms traverse only a fraction of the search space. The trade-off between effort and result quality is governed by two parameters: the time limit τ and the level of parallelism k.

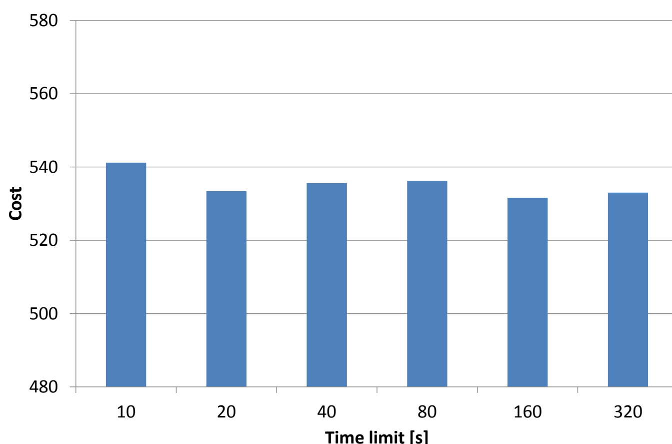


Fig. 9. Cost of the solution delivered by the CP algorithm with different values of the time limit τ

4. Recall that n denotes the number of VMs, m the number of PMs.

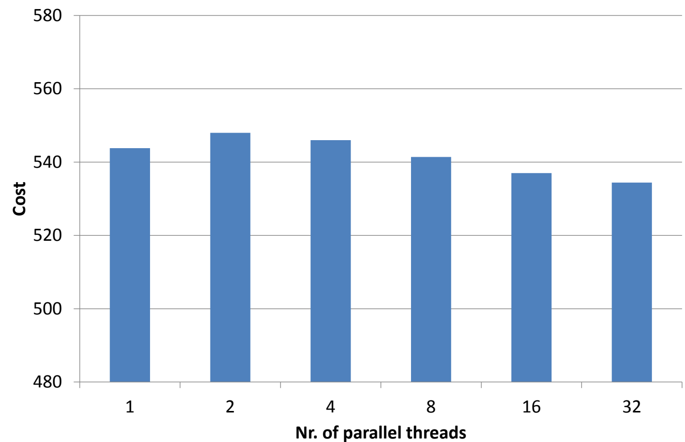


Fig. 10. Cost of the solution delivered by the CP algorithm with different number of parallel search threads (k)

In this experiment, the number of PMs is fixed at 200 and the number of VMs at 400, and the results of the CP algorithm are shown for varying values of τ (Fig. 9) and k (Fig. 10).

As expected, increasing either τ or k typically leads to results with lower costs, although this is not always the case: since the algorithm is randomized, different runs may explore different parts of the search space, leading to some noise in the results.

Interestingly, increasing k has a more pronounced effect on the cost of the result than increasing τ. This can be seen also quantitatively: the Pearson correlation coefficient between k and the cost of the result is -0.91, whereas between τ and the cost of the result it is -0.56. They are both negative, meaning that an increase in k or τ tends to lead to a decrease in solution cost, but a value near to -1 indicates a stronger correlation. Moreover, increasing k from 1 to 32 decreases solution cost by 1.7%, whereas increasing τ from 10s to 320s decreases solution cost by only 1.5%.

This difference is probably due to the fact that running the same search for more time leads to more thorough exploration of the same part of the search space, which may actually be far from the optimum, whereas running more searches increases the probability that better regions of the search space are reached. An interesting consequence is that even if there are not sufficient parallel processing units available, it may be better to make several shorter searches sequentially than making a single long search. This finding is in line with previous experience with other combinatorial algorithms for hard problems [30].

It can also be observed that the effect of increasing both k and τ is rather small. This means that already modest computation time and resources are sufficient for the algorithm to perform quite well.

6.5 Experiment 4: cost function

Our cost function is the weighted sum of the number of active PMs, the number of migrations, and the number

of core overloads. Now we investigate how different settings for the weights influence the tradeoff that the CP algorithm finds between the conflicting optimization goals.

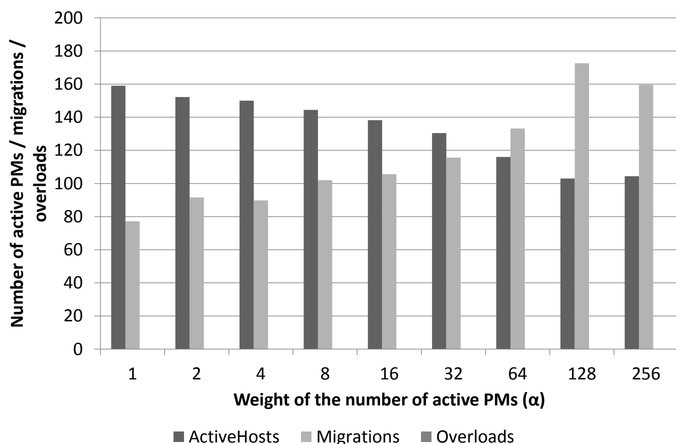


Fig. 11. Effect of changing the weight of the number of active PMs in the cost function (α)

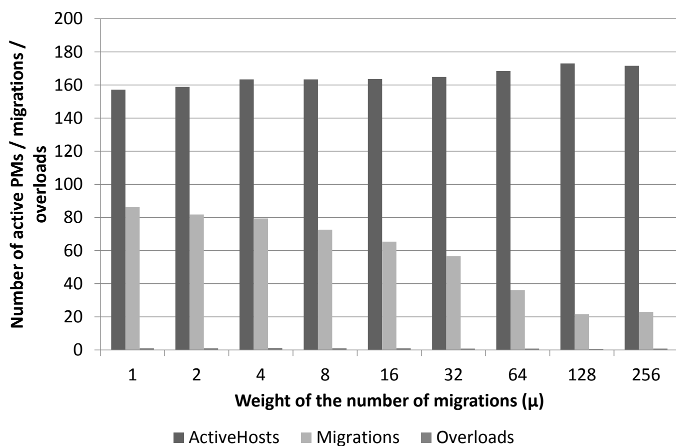


Fig. 12. Effect of changing the weight of the number of migrations in the cost function (μ)

In Fig. 11, the weight of the number of active PMs is varied from 1 to 256, while the other two weights are set to 1. As expected, increasing α leads to solutions with fewer active PMs, at the cost of an increase in the number of migrations. Similarly, in Fig. 12, the weight of the number of migrations is varied from 1 to 256, while the other two weights are set to 1. This leads to solutions with fewer migrations, at the cost of a higher number of active PMs. Thus we can conclude that by setting the weights appropriately, the trade-off between these two conflicting goals can be tuned.

It is interesting to note that the number pCPU overloads is always very small, in many cases even 0. This is probably due to the proprietary constraint that we implemented in order to make the calculation of the number of pCPU overloads more efficient (see Section 5.4). This constraint offers stronger pruning capabilities than the built-in constraints used for calculating the other two cost components, introducing a bias towards solutions with a very low number of pCPU overloads.

6.6 Experiment 5: real-world trace

After the experiments with synthetic test data, we also wanted to explore the applicability of our algorithms in a more realistic scenario. For this purpose, we used the fastStorage trace from the Bitbrains data center, serving enterprise applications mainly in the financial domain, which is one of the very few publicly available virtualized IaaS traces [36]. This trace, available from the Grid Workloads Archive[5], contains resource usage data from 1,250 VMs, including CPU load (in terms of the provisioned CPU capacity in MHz), sampled every 5 minutes. The VMs have 1 to 32 cores, with an average of 3.3 vCPUs per VM. The evolution of the overall compute demand of the VMs over time is shown in Fig. 13.

For our experiment, we used the trace data from 24 hours, i.e., 288 consecutive samples. For evaluating one of our algorithms, we let it compute a new mapping after each sample, based on the VM sizes from the sample and the mapping that it had computed previously, thus re-optimizing the mapping after every 5 minutes.

Unfortunately, the public trace does not contain information about the underlying hardware resources. Therefore, we assume the following hardware configuration:

- 150 PMs with Intel Xeon X5570 CPUs, with 8 pCPUs running at 2.93GHz
- 150 PMs with Intel Xeon E5530 CPUs, with 4 pCPUs running at 2.40GHz

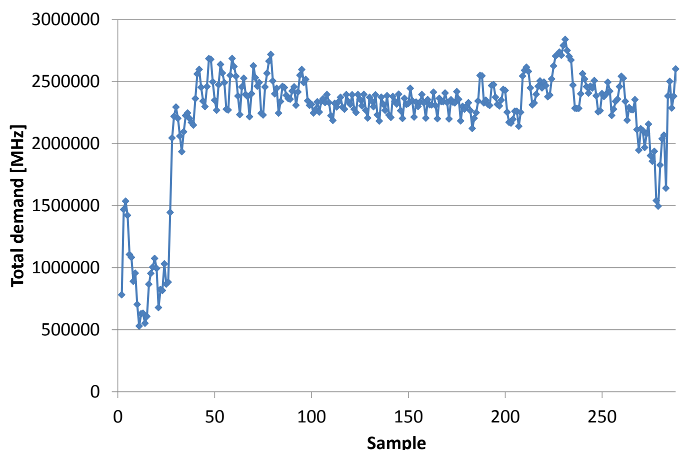


Fig. 13. Evolution of the demand of the 1,250 VMs in the Bitbrains trace over 288 consecutive samples (24 hours)

Although the number of PMs (300) and VMs (1,250) is not significantly greater than in our previous experiments, the average number of cores per machine is considerably higher in this case. Since this makes the total number of pCPUs and vCPUs quite high, only the greedy, hybrid1, hybrid2, and hybrid3 algorithms can be

5. http://gwa.ewi.tudelft.nl

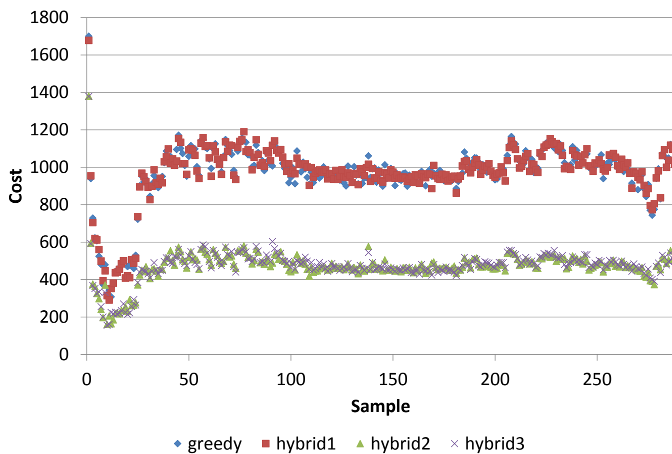


Fig. 14. Results of different VM placement algorithms on the Bitbrains trace

applied (the systematic global search procedures of the cp and twostage algorithms do not produce valid results within the given time limit). The results are shown in Fig. 14.

As can be seen, the results of all algorithms closely follow the evolution of the demand. However, the hybrid2 and hybrid3 algorithms perform consistently much better than the other two (just like in the previous experiments). The best algorithm outperforms the non-multicore-aware greedy algorithm by roughly 50%.

7 Conclusions and Future Work

In this paper we argued that ignoring the scheduling of cores during VM placement is an over-simplification that may lead to suboptimal results, and we showed that core-level placement information is necessary in many cases to achieve good overall performance and cost. We presented a possible formulation of the VM placement problem, in which pCPUs can be shared in non-trivial ways between vCPUs. We proposed constraint programming to cope with the resulting complex optimization problem, and also several possible heuristically boosted variants of the pure CP approach. Our empirical results showed that the new algorithms outperform a traditional non-multicore-aware approach by 25-60%. Pure CP delivers excellent results within acceptable time for up to 350 PMs and 700 VMs, whereas the hybrid algorithms produce very good results even for 1000 PMs and 2000 VMs. Thus we can conclude that the combined problem of VM placement and core scheduling can be effectively approached with the presented methods for practically useful problem sizes.

As future research, we would like to extend the presented approach with other resource dimensions (like memory and I/O). We expect this to further constrain the search space, which is advantageous for the CP-based approaches. Furthermore, we would like to enhance the presented systematic search algorithms with more advanced heuristics for narrowing down the search space.

Acknowledgments

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

References

[1] J. Ahn, C. Kim, J. Han, Y. ri Choi, and J. Huh, “Dynamic virtual machine scheduling in clouds for architectural shared resources,” in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing (HotCloud’12)*, 2012.

[2] A. M. Al-Qawasmeh, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, “Power and thermal-aware workload allocation in heterogeneous data centers,” *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 477–491, 2015.

[3] 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)*, 2007, pp. 209–213.

[4] 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*, vol. 28, pp. 755–768, 2012.

[5] 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*, 2010, pp. 577–578.

[6] ——, “Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers,” *Concurrency and Computation: Practice and Experience*, vol. 24, no. 13, pp. 1397–1420, 2012.

[7] ——, “Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, pp. 1366–1379, 2013.

[8] L. F. Bittencourt, E. R. Madeira, and N. L. da Fonseca, “Scheduling in hybrid clouds,” *IEEE Communications Magazine*, vol. 50, no. 9, pp. 42–47, 2012.

[9] 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*, 2007, pp. 119–128.

[10] A. Bockmayr and J. Hooker, “Constraint programming,” in *Handbook of Discrete Optimization*, K. Aardal, G. Nemhauser, and R. Weismantel, Eds., 2005, pp. 559–600.

[11] R. 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*, 2010, pp. 228–235.

[12] 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*, 2011, pp. 161–168.

[13] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility,” *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[14] M. Carlsson, “SICStus Prolog user’s manual, release 4.2.3,” http://sicstus.sics.se/sicstus/docs/4.2.3/pdf/sicstus.pdf, 2012.

[15] M. Carlsson, G. Ottosson, and B. Carlson, “An open-ended finite domain constraint solver,” in *Programming Languages: Implementations, Logics, and Programs*, 1997, pp. 191–206.

[16] M. DeVuyst, A. Venkat, and D. M. Tullsen, “Execution migration in a heterogeneous-ISA chip multiprocessor,” in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 261–272.

[17] C. Dupont, G. Giuliani, F. Hermenier, T. Schulze, and A. Somov, “An energy aware framework for virtual machine placement in cloud federated data centres,” in *Third International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy)*, 2012.

[18] 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*, vol. 79, pp. 1230–1242, 2013.

[19] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, “Resource pool management: Reactive versus proactive or let’s be friends,” *Computer Networks*, vol. 53, no. 17, pp. 2905–2922, 2009.

[20] 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*. Springer, 2012, pp. 81–92.

[21] B. Guenter, N. Jain, and C. Williams, “Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning,” in *Proceedings of IEEE INFOCOM*. IEEE, 2011, pp. 1332–1340.

[22] F. Hermenier, J. Lawall, and G. Muller, “BtrPlace: A flexible consolidation manager for highly available applications,” *IEEE Transactions on Dependable and Secure Computing*, vol. 5, pp. 273–286, 2013.

[23] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, “Entropy: a consolidation manager for clusters,” in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2009, pp. 41–50.

[24] E. Jeannot, G. Mercier, and F. Tessier, “Process placement in multicore clusters: Algorithmic issues and practical techniques,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 4, pp. 993–1002, 2014.

[25] 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*, 2010, pp. 62–73.

[26] 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 Parallel Processing*. Springer, 2013, pp. 317–328.

[27] D. Kim, H. Kim, and J. Huh, “Virtual snooping: Filtering snoops in virtualized multi-cores,” in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 459–470.

[28] D. Lago, E. Madeira, and L. 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.

[29] 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)*. Springer, 2011, pp. 120–134.

[30] Z. A. Mann, *Optimization in computer engineering – Theory and applications*. Scientific Research Publishing, 2011.

[31] ——, “Allocation of virtual machines in cloud data centers – a survey of problem models and optimization algorithms,” *ACM Computing Surveys*, vol. 48, no. 1, 2015.

[32] ——, “Rigorous results on the effectiveness of some heuristics for the consolidation of virtual machines in a cloud data center,” *Future Generation Computer Systems*, vol. 51, pp. 1–6, 2015.

[33] 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.

[34] B. C. Ribas, R. M. Suguimoto, R. A. N. R. Montano, F. Silva, L. de Bona, and M. A. Castilho, “On modelling virtual machine consolidation to pseudo-Boolean constraints,” in *13th Ibero-American Conference on AI*, 2012, pp. 361–370.

[35] 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*. IEEE, 2012, pp. 844–851.

[36] S. Shen, V. van Beek, and A. Iosup, “Statistical characterization of business-critical workloads hosted in cloud datacenters,” in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 465–474.

[37] 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*, 2013, pp. 9–15.

[38] K. S. Shim, M. Lis, O. Khan, and S. Devadas, “Thread migration prediction for distributed shared caches,” *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 53–56, 2014.

[39] S. Srikantaiah, A. Kansal, and F. Zhao, “Energy aware consolidation for cloud computing,” *Cluster Computing*, vol. 12, pp. 1–15, 2009.

[40] A. Strunk, “Costs of virtual machine live migration: A survey,” in *8th IEEE World Congress on Services*, 2012, pp. 323–329.

[41] P. Svärd, W. Li, E. Wadbro, J. Tordsson, and E. Elmroth, “Continuous datacenter consolidation,” Umea University, Tech. Rep., 2014.

[42] L. Tomás and J. Tordsson, “An autonomic approach to risk-aware data center overbooking,” *IEEE Transactions on Cloud Computing*, vol. 2, no. 3, pp. 292–305, 2014.

[43] E. Tsamoura, A. Gounaris, and K. Tsichlas, “Multi-objective optimization of data flows in a multi-cloud environment,” in *Proceedings of the Second Workshop on Data Analytics in the Cloud*, 2013, pp. 6–10.

[44] A. Verma, P. Ahuja, and A. Neogi, “pMapper: power and migration cost aware application placement in virtualized systems,” in *Middleware 2008*, 2008, pp. 243–264.

[45] 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*, 2009, pp. 355–368.

[46] VMware, Inc., “Understanding Oracle certification, support and licensing for VMware environments,” http://www.vmware.com/files/pdf/techpaper/vmw-understanding-oracle-certification-supportlicensing-environments.pdf, 2015.

[47] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, “Sandpiper: Black-box and gray-box resource management for virtual machines,” *Computer Networks*, vol. 53, no. 17, pp. 2923–2938, 2009.

[48] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova, “1000 islands: an integrated approach to resource management for virtualized data centers,” *Cluster Computing*, vol. 12, no. 1, pp. 45–57, 2009.

Zoltán Ádám Mann received the MSc and PhD degrees in Computer Science from Budapest University of Technology and Economics in 2001 and 2005, respectively. He is currently associate professor at the Department of Computer Science and Information Theory at Budapest University of Technology and Economics. His research interests include optimization problems and algorithms in cloud computing.

Multicore-aware virtual machine placement in cloud data centers

—Supplemental material—

Zoltán Ádám Mann

A Multicore scheduling issues

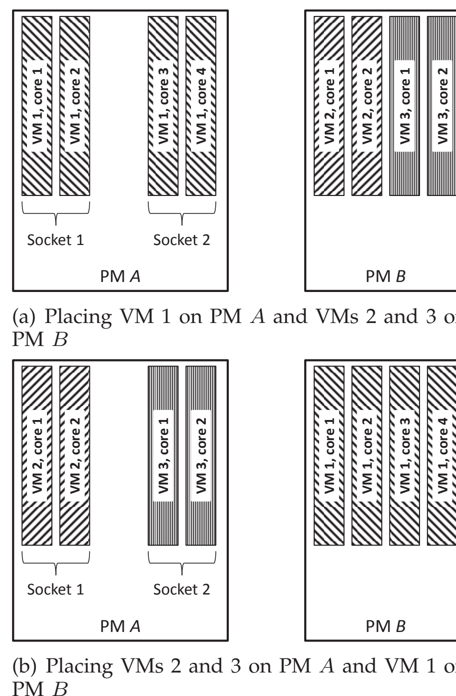
Here, some extensions to Section 3 of the paper are discussed.

A.1 Multi-socket and NUMA architectures

The effects of the vCPU-pCPU mapping on VM performance are considerably amplified by multi-socket systems: if the vCPUs of a VM are mapped to pCPUs in different sockets, this can degrade the performance of the VM significantly. For example, Ibrahim et al. report an average performance degradation of 20% for CPU-intensive VMs on a 4-socket machine [7]. The situation is even worse for NUMA (non-uniform memory access) architectures because of the loss of data locality through the mapping on multiple NUMA nodes. In the same article, performance degradation of up to 82% is reported for a 4-node NUMA machine [7].

A VM placement algorithm that ignores the CPU structure of the PMs may lead to large performance penalties. For example, consider two PMs and three VMs: PM A has two dual-core CPU sockets whereas PM B has a single quad-core CPU socket; VM 1 has four cores and VMs 2 and 3 have two cores each. Each pCPU has a capacity of 1000 MIPS and each vCPU requires also 1000 MIPS. As shown in Fig. 1, there are two possible VM placements, and for a CPU-core-oblivious VM placement algorithm, they seem to be equally good. However, the placement in Fig. 1(a) would incur a significant penalty because of the lack of containment for VM 1.

Seemingly, this issue could be remedied if NUMA nodes are modeled as multiple PMs for the VM placement algorithm. But this is not a good solution because the NUMA nodes of a machine can share important resources (e.g., disk or network interface) that a VM placement algorithm should take into account.



(a) Placing VM 1 on PM A and VMs 2 and 3 on PM B

(b) Placing VMs 2 and 3 on PM A and VM 1 on PM B

Fig. 1. Two possible VM placements

A.2 Effects of hyper-threading

Many of today's servers use processors that support simultaneous multi-threading, often called hyper-threading (HT). With HT, a pCPU can run two threads in parallel. This usually results in better use of the available resources because if one thread must wait for a load instruction, the other can still perform useful work. A pCPU with HT enabled appears as two "logical" cores.

However, because of the shared resources, the resulting two logical cores do not offer twice the performance of a non-HT core. According to Intel's original estimate, HT may result in performance improvement of up to 30% [14], which is consistent with recent independent measurements [15], [19]. The actual performance improvement depends on the characteristics of the workload, such as memory access patterns and inter-thread communication patterns [16].

A VM placement algorithm may regard a hyper-

threaded core as one or two pCPUs, but the capacity of the pCPUs is different in the two cases. For example, a core with non-HT performance of 1000 MIPS may offer HT performance of 1300 MIPS. Thus it can be regarded either as a pCPU with 1000 MIPS or two pCPUs with 650 MIPS each. Which one of the two is better depends on the kinds of VMs that need to be allocated: for example, for allocating a VM with a single 800 MIPS vCPU, the first case is better, whereas for allocating a VM with two 600 MIPS vCPUs, the second is better. Unfortunately, a non-multicore-aware VM placement algorithm will not be able to model this situation correctly.

A.3 Dedicated cores

Although virtualization ensures some level of isolation between co-located VMs, this isolation is not perfect: e.g., the shared last-level cache and memory interface may lead to contention between co-located VMs, thus potentially resulting in performance degradation [9]. Known as the "noisy neighbor" effect, a VM exerting high pressure on the shared resources may seriously degrade the performance of another VM on the same PM. Kocsis et al. showed that a 10-second CPU burst of one VM may lead to an outage of several minutes for a performance-sensitive application running in a co-located VM [11].

For performance-critical VMs (e.g., soft real-time applications), therefore, it is good practice to allocate dedicated vCPUs, thus minimizing the noisy neighbor effect [10]. However, this is a tricky situation for a VM placement algorithm that only considers the total CPU capacity of PMs and the total CPU load of VMs. The problem is that the CPU capacity that a given VM occupies from a PM is not constant but depends on the PM. For example, consider a single-core VM with CPU load 500 MIPS, requiring a dedicated core. If the VM is allocated on a PM whose cores have 1000 MIPS capacity, then the VM occupies 1000 MIPS from this PM. However, on a PM with 2000-MIPS cores, the same VM occupies 2000 MIPS, a situation not foreseen by current VM placement algorithms.

A.4 Asymmetric processors

Many of the published VM placement algorithms assume that all PMs are equal, see e.g. [2], [3], [4], [5], [17], [18], [20]. More advanced algorithms take into account that PMs may be different in terms of capacity and/or energy efficiency [1], [8], [13].

Heterogeneity is also possible within a PM. This is the case for asymmetric multicore chips, which feature cores of different computational capacity and power efficiency. It has been shown that for several types of workloads, asymmetric processors can achieve better performance and power characteristics than symmetric CPUs in which all cores are the same [6], [12].

To take advantage of asymmetric processors, VM placement must be aware of the per-core capabilities of the PMs. For example, if a PM possesses one fast core and several slower cores, then it will be a good host for a single-core VM with high computational load and some further VMs with low load, but it is not suitable for a VM requiring two high-performance vCPUs.

B Connection of the problem formulation to the above issues

It is worth to revisit the issues presented in Section 3 of the paper and above and discuss how they can be addressed in the framework of our problem formulation (presented in Section 4 of the paper).

- **High sequential compute demand**. If a vCPU is mapped to a pCPU with lower capacity, this will automatically lead to a pCPU overload. Since the problem formulation aims at minimizing the number of pCPU overloads, it will aim at eliminating such situations as much as possible.
- **vCPU migration vs. pinning**. This has already been discussed in the paper.
- **Multi-socket and NUMA architectures**. Our problem model contains two levels: machines (PM/VM) and cores (pCPU/vCPU). In order to accurately model multi-socket architectures, three levels would be needed: machines, sockets, and cores. A less precise but much simpler alternative is to use just two levels: machines and sockets, and treat the cores belonging to the same socket as one big core with the total capacity of those cores. This is a sensible model if vCPU movement is much cheaper within a socket than between different sockets. This is then the same as our model, with sockets in lieu of cores.
- **Effects of hyper-threading**. In its current form, the problem model does not address HT issues. However, the availability of per-core information ($pcc(p)$ and $vcl(vc)$) and the core mapping ($cmap$) makes it relatively easy to extend the problem model with HT. A HT-capable core can be modeled with a pair of pCPUs pc_1, pc_2 so that either pc_1 has capacity pcc_{non-HT} and pc_2 has capacity 0, or both have capacity pcc_{HT}. The definition of when a core is overloaded must be changed accordingly.
- **Dedicated cores**. Again, the availability of core mapping information makes it easy to incorporate this in the problem model. If VM v requires dedicated cores, then for each vCPU vc of v, the following must be ensured: $|cmap^{-1}(cmap(vc))| = 1$.
- **Asymmetric processors**. It is straight-forward to extend the model to incorporate this: instead of the PM-level $pcc(p)$ numbers, each pCPU pc may have its own $pcc(pc)$ capacity.

References

[1] 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*, 2011, pp. 161–168.

[2] ——, “Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds,” in *Proceedings of IEEE Infocom 2012*, 2012, pp. 2861–2865.

[3] 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*, vol. 79, pp. 1230–1242, 2013.

[4] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, “Resource pool management: Reactive versus proactive or let’s be friends,” *Computer Networks*, vol. 53, no. 17, pp. 2905–2922, 2009.

[5] S. He, L. Guo, M. Ghanem, and Y. Guo, “Improving resource utilisation in the cloud environment using multivariate probabilistic models,” in *IEEE 5th International Conference on Cloud Computing*, 2012, pp. 574–581.

[6] M. D. Hill and M. R. Marty, “Amdahl’s law in the multicore era,” *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[7] K. Z. Ibrahim, S. Hofmeyr, and C. Iancu, “The case for partitioning virtual machines on multicore architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2683–2696, 2014.

[8] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whalley, and E. Snible, “Improving performance and availability of services hosted on IaaS clouds with structural constraint-aware virtual machine placement,” in *IEEE International Conference on Services Computing (SCC)*, 2011, pp. 72–79.

[9] S.-g. Kim, H. Eom, and H. Y. Yeom, “Virtual machine consolidation based on interference modeling,” *The Journal of Supercomputing*, vol. 66, no. 3, pp. 1489–1506, 2013.

[10] I. Kocsis, Z. A. Mann, and D. Zilahi, “Optimal deployment for critical applications in infrastructure as a service,” in *3rd International IBM Cloud Academy Conference (ICACON 2015)*, 2015.

[11] I. Kocsis, A. Pataricza, Z. Micskei, A. Kövi, and Z. Kocsis, “Analytics of resource transients in cloud-based applications,” *International Journal of Cloud Computing*, vol. 2, no. 2-3, pp. 191–212, 2013.

[12] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, “Heterogeneous chip multiprocessors,” *Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[13] D. Lago, E. Madeira, and L. 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.

[14] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton, “Hyper-threading technology architecture and microarchitecture,” *Intel Technology Journal*, vol. 6, pp. 4–15, 2002.

[15] C. McCreesh and P. Prosser, “Multi-threading a state-of-the-art maximum clique algorithm,” *Algorithms*, vol. 6, pp. 618–635, 2013.

[16] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas, “The impact of hyper-threading on processor resource utilization in production applications,” in *18th International Conference on High Performance Computing (HiPC)*, 2011, pp. 1–10.

[17] S. Srikantaiah, A. Kansal, and F. Zhao, “Energy aware consolidation for cloud computing,” *Cluster Computing*, vol. 12, pp. 1–15, 2009.

[18] P. Svärd, W. Li, E. Wadbro, J. Tordsson, and E. Elmroth, “Continuous datacenter consolidation,” Umea University, Tech. Rep., 2014.

[19] I. Zecena, M. Burtscher, T. Jin, and Z. Zong, “Evaluating the performance and energy efficiency of n-body codes on multicore CPUs and GPUs,” in *IEEE 32nd International Performance Computing and Communications Conference*, 2013, pp. 1–8.

[20] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova, “1000 islands: an integrated approach to resource management for virtualized data centers,” *Cluster Computing*, vol. 12, no. 1, pp. 45–57, 2009.