Leveraging vCPU-Utilization Rates to Select Cost-Efficient VMs for Parallel Workloads

William F. C. Tavares
wtavares@lmcad.ic.unicamp.br
University of Campinas
Campinas, São Paulo, Brazil

Marcio R. M. Assis
marcio.miranda.assis@gmail.com
University of Campinas
Campinas, São Paulo, Brazil

Edson Borin
edson@ic.unicamp.br
University of Campinas
Campinas, São Paulo, Brazil

ABSTRACT

The increasing use of cloud computing for parallel workloads involves, among many problems, resources wastage. When the application does not fully utilize the provisioned resource, the end-of-the-month bill is unnecessarily increased. This is mainly caused by the user's inexperience and naïve behavior. Many studies have attempted to solve this problem by searching for the optimal VM flavor for specific applications with specific inputs. However, most of these solutions require knowledge about the application or require the application's execution on multiple VM flavors. In this work, we propose four new heuristics that recommend cost-effective VMs for parallel workloads based solely on the vCPU-utilization rate of the currently executing VM flavor. We also evaluate them on two scenarios and show that the core-heuristic is capable of recommending VM flavors that have minimal impact on performance and reduce the applications cost, on average, by 1.5× (3.0×) on high (low) vCPU-utilization rate scenarios.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **General and reference** → **Performance**.

KEYWORDS

Cloud computing, Cost optimization, Resource utilization

ACM Reference Format:
William F. C. Tavares, Marcio R. M. Assis, and Edson Borin. 2021. Leveraging vCPU-Utilization Rates to Select Cost-Efficient VMs for Parallel Workloads. In *2021 IEEE/ACM 14th International Conference on Utility andCloud Computing (UCC'21), December 6–9, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3468737.3494095

1 INTRODUCTION

Cloud computing has become an alternative to cluster computing in relation to high-performance computing resources, mainly due to the advantages of high accessibility and flexibility and benefits such as low deployment time and acquisition and maintenance cost. Additionally, the variety of available computing resources and the ability to scale and dynamically provision them have caught the attention of the community that works with parallel algorithms and High-Performance Computing (HPC) applications [19]. These advantages are compromised when the utilization cost are unnecessarily high and avoidable. Naïve user behavior seems to be at fault for issues that lead to mistakes when choosing the resources. These not only increase the bill at the end of the month, but often degrade the application's performance.

The ideal scenario is to provision the exact amount of resources that the application needs. However, this amount is not always known by the user – who may not be expert on the application – or hard to measure – due to the varying execution behavior with different inputs or due to randomness of the application algorithms [5]. This becomes more complex on public cloud providers, who offer hundreds of VM configurations [30]. Additionally, in many cases, applications are not prepared to be executed in the cloud, and need to be adapted to deal with the cloud dynamics and shared environment in order to effectively use its resources [15].

From utilization data in Google's clusters, Reiss *et al.* [21] showed that the users tend to overprovision their resources: 70% of processing resources in high priorities nodes were allocated, but only 20% of them were being used. Situations such as these have become one of the main concerns of enterprises, as shown by a 2019 estimate that indicates a wastage of $5.3 billions of dollars due to cloud resource overprovisioning [7].

Some studies use the overprovision behaviour to optimize the resources utilization from the cloud provider's point of view for overbooking [27] or energy consumption optimization [17]. In the user's perspective, studies attempt to create elastic applications to get the most out of cloud elasticity, which requires code refactoring [15]. To avoid refactoring the code, another solution is to detect overprovision and select a better resource configuration. Some of the studies focus on specific types of applications, such as *embarrassingly parallel* [24], MapReduce [13], and *data-intensive workflow* [20] applications, or require additional time and cost to operate, as they use complex techniques (like machine learning and Bayesian Optimization) to model the application's performance [1, 28, 30].

In this work, we address the scenarios where users are migrating their parallel applications to the cloud but do not have enough experience to properly select the best computing resources for their applications. To this end, we propose a solution to avoid the overprovision by finding a VM flavor that optimizes the cost by only knowing the CPU utilization in the VM. As this work focuses on parallel workloads, we are also concerned with maintaining the performance, while optimizing the cost.

The remaining of the text is organized as follows. Section 2 describes the problem of migrating applications to the cloud, formulate the problem, and present related works. Section 3 presents four new heuristics to select VM flavors that optimizes the cost

UCC'21, December 6–9,2021, Leicester, United Kingdom

ACM ISBN 978-1-4503-8564-0/21/12…$15.00
https://doi.org/10.1145/3468737.3494095

without compromising the performance. Section 4 details the experiments and presents the results. Finally, Section 5 presents our conclusions and future work.

2 BACKGROUND

In the Infrastructure as a Service (IaaS) model, cloud providers offer a large variety of computing resources, such as virtual machines, that can be instantiated and accessed through the Internet, and the user is charged by time of usage. These resources follow a predefined configuration, known as *flavor*, which defines the hardware configuration, the capacity of the resources and the price.

As HPC users are becoming more interested in the cloud, more parallel applications are being executed in it. A common strategy to migrate applications to the cloud is by copy and pasting it [15], which consists of executing the application without modification. This is more susceptible to resource wastage as these applications are not designed to deal with the cloud, which is a virtualized environment with multiple tenants. The bad usage of cloud resources is reinforced by the naïve behavior and inexperience of new users that are susceptible to selecting costly flavors for their applications. A consequence of it is the **underutilization**, defined by Espadas *et al.* [8] as an effect that happens "whenever some resources are not being used by virtual machines within a cloud computing infrastructure, and an application is being executed". As a consequence, comes the **overprovision**, which is the provision of computing resources that are not entirely used by the application. This results in computing resource wastage, as they are being underutilized. By reducing the resource wastage, the operational cost is also reduced.

The objective of this work is to optimize the usage cost of parallel workloads on the cloud without compromising their performance. We approach this problem by detecting and minimizing the overprovision effect.

2.1 Problem formulation

Given a set of computing resources $R = \{r_1, ..., r_n\}$, $n \in \mathbb{N}$, and a set of flavors of virtual machines $F = \{f_1, ..., f_m\}$, $m \in \mathbb{N}$, each flavor is defined by a tuple $f_k = (C, p)$, $k \in \{1, ..., m\}$, where $C = \{C_{r_1}, ..., C_{r_n}\}$ describes the capacity of each resource in the flavor and $p \in \mathbb{R}^+$ is the price per time-unit.

Given a virtual machine V with flavor f_k, the utilization $u_r, \forall r \in R$, is defined as the percentage of the capacity C_r that is being utilized by the application, so $0\% \leq u_r \leq 100\%$. V has a cost attached to it, defined as $cost(V) = p \times t$, where t is the time the virtual machine is being used, in the same time-unit as the price.

The overprovision happens when $\exists r \in R \mid u_r < 100\%$ in V. As we focus on optimizing the service's cost, we seek to select a flavor $f_s = (\hat{C}, \hat{p})$ in which the cost of usage is reduced, $cost(\hat{V}) < cost(V)$, $\hat{V}$ is V using flavor f_s. As our constraint is to maintain the performance, the selection must consider the resource capacities in a way to not degrade the performance. The capacity utilization of each resource in V that does not cause underutilization must be sustained, this is, $\hat{C}_r \equiv C_r \mid C_r = 100\% \; \forall r \in R$, while the capacity utilization of each resource in V that causes underutilization may be reduced, so, $\hat{C}_r < C_r \mid C_r \neq 100\% \; \forall r \in R$.

2.2 Related works

There are two main approaches to detect overprovision: reactive and predictive [10]. The reactive approach is based on Rule-Condition-Action mechanisms, where there is a rule composed of a set of conditions that, when satisfied, triggers actions; this technique controls the elasticity in runtime, but the event of interest needs to happen before the reaction. On the other hand, the predictive approach rely on heuristics, mathematical analysis, or machine learning to predict the application's behavior and prevent the event from happening, however, the workload needs to be predictable.

There are several works that search for methodologies to avoid overprovision using the predictive approach. Lloyd *et al.* [18] investigated a method to determine the amount of VMs of a given flavor required to match the performance of another set of VMs of different flavors using only the resource utilization rate. Their method requires pre-executions on each flavor to be able to create the equivalence model. Kenga *et al.* [16] developed a methodology to determine the exact amount of resources to minimize the overprovision for a specific application execution using previous data. Sharma *et al.* [23] proposed a solution that considers an estimated peak workload that the VM must sustain. The methodology uses a perfect forecaster that knows the workload in advance. Given the peak, a cost minimization is performed using integer linear programming (ILP) and heuristics, discovering the least-costly combination of flavors that can compute this peak. Jung *et al.* [14] use a combination of predictive models and graph search techniques to develop a solution for multi-tier applications that are associated with transactions – through which users access its services.

Some works focus on predictive models for workloads schedulers for different type of applications, such as *workflow based* [6, 25], *embarrassingly parallel* [24], and *MapReduce bases* [13].

Other works focus on selecting the optimal flavor for the application in terms of cost. By using the optimal flavor, the cost is already optimized, so the underutilization is tolerable, if it exists. Ernest [28] predicts the performance on a specific configuration, by modeling the performance with samples of the original input; this approach considers the application that have linear (or quasi-linear) correlation between execution time and input size. This performance model works only for a specific flavor, needing to execute in other flavors to estimate the performance on them. Cherrypick [1] finds optimal or near-optimal configuration that optimizes the cost and guarantees the application's performance, by estimating the confidence interval in both cost and performance configuration using Bayesian Optimization (BO). During the search, it tells which configuration should be sampled next to find the optimal one, this avoids testing all configurations. Although this solution focus on big data analytic applications, it avoids specific insights and only use attributes about the flavor (such as CPU amount and memory capacity). Hsu *et al.* [11] identified a limitation when applying the BO method in specific workloads and attributed this to the insufficient information that number of cores and memory capacity have to predict the application's behavior. They proposed a method that use low-level metrics (such as I/O wait and memory usage) from previous measurements to estimate the performance in the same flavor.

PARIS [30] enables the user to tolerate mild reductions in performance for substantial cost savings by estimating the trade-off between performance and cost for all VM flavors. It is divided in two steps: offline and online. In the offline step, experiments with benchmarks are executed to make a correlation between the flavors and the resource utilization. In the online step, a representative job of the application is executed on a pair of reference flavors and the results are compared with all flavors considering the correlation.

Micky [12] proposes a methodology for multiple workloads by formulating the problem of searching the optimal VM as a multi-armed bandit problem. SARA [29] comes to circumvent the problem of Micky, that present bad results for small group of workloads, and can stably and quickly optimize any amount of workloads, by clustering the workloads and applying Deep Reinforcement Learning, then, a new workload is clustered and uses the optimal configuration of its cluster.

Baughman *et al.* [3] explore two approaches to measure and predict performance, one with prior knowledge of the application and other with prior execution history in non-cloud environment. Some works focus on frameworks such as Hadoop [2] and Spark [9], and multiple frameworks [4].

Brunetta and Borin [5] propose a methodology that performs a short-running of the application on a set of configurations, however, it requires executing the application on multiple flavors and depend on the correlation with the full-running to be able to predict the performance in each configuration.

In general, solutions that use the predictive technique depend on pre-executions or history data, meaning that comes with an operation cost, and depend on the application being likely to have its behavior predicted.

Different from the others, the solution proposed by Sedaghat *et al.* [22] is reactive. They consider a distributed application running in a set of VMs and, when the overprovision is detected, the solution searches within the flavors' set for a new optimal set of VMs that matches the capacity needed using ILP. The reconfiguration to the new set of VMs is evaluated, as it is not always feasible and can come with higher cost – depending on the application and its duration. This evaluation uses an equation that considers the old and new set of VMs and the reconfiguration cost. This equation depends on the expected duration of the new optimal set; on the data of one metric, such as CPU utilization; and a gain factor, which depends on the experience of the user.

Public cloud providers offer services to optimize the provisioning cost. There are reactive services that automate the VM horizontal elasticity[1,2] and the vertical scaling[3,4]. There are also *right-sizing* services, which search for a match between the flavor's size, the workload, and capacity requirements using predictive methods based on machine learning[5,6]. The main issue, besides the additional cost, resides on the provider dependency, which restrains the use of these services in other providers. Most services use data from its provider monitoring tool, which comes with the limitations of these tools and their cost. The user must also trust the provider, as they are responsible for implementing and auditing these services.

In our problem, the objective is to optimize the cost of parallel workloads by reducing the resource wastage while maintaining the performance. One of the main concerns of solutions that optimize cost is the time and cost to operate, for this reason, we opted for a reactive solution, to avoid modeling the performance with complex techniques or multiple pre-executions in the cloud. Different from Sedaghat *et al.* [22]'s work, which is also reactive, our solution does not depend on the user's experience. We start from the premise that the user is not knowledgeable about the cloud and/or the application and migrated using the *copy & paste* technique. In this way, we cannot require users to modify their applications nor expect them to have insights about their behavior. Furthermore, as our solution tends to maintain the performance, each node can be cost-optimized individually, as long as the application's performance depends only on the resources of this node and not the others.

3 SELECTING OPTIMAL VM FLAVORS

Our solution aims at identifying virtual machine flavors that optimize the cost of parallel workloads in execution without compromising their performance based on their resource's utilization data.The utilization data can be gathered by open-source tools (Zabbix [7], Prometheus [8]) or provider's tools (CloudWatch [9] from Amazon Web Services (AWS)). With the utilization data of an application execution, our solution can be used by automatic cloud resource management tools to make recommendations and/or take actions to prevent wastage. For example, this information may be used by the Bucket Solution [26], a methodology that depends on an algorithm to identify the optimal flavor for a given application in order to quantify the wastage caused by its execution.

Algorithm 1 presents our solution. It is divided in two steps: filtering and searching. The first step (line 1) filters all the flavors that do not offer the minimum amount of each resource and, at the end, produces a set of candidate flavors. The second step (lines 2-7) takes these candidates and searches for the optimal one using the vCPU utilization rate. For this last step, we propose four heuristics.

Algorithm 1: Selection heuristics

input : The virtual machine V, the available $flavors$
output: The optimal flavor

1 $candidates \leftarrow \text{filter}(V, flavors)$
2 $effective \leftarrow \frac{\lambda_{vcpu,\Lambda}.u_{vCPU} \times V.flavor.C_{vcpu}}{100}$
3 $amount \leftarrow \text{getSearchList}(candidates, effective)$
  $amount \leftarrow \text{getSearchList}(candidates, 2 \times effective)$
4 $cheapest \leftarrow \text{getCheapest}(candidates, amount)$
5 **if** $cheapest.price < V.price$ **then**
6 | **return** $cheapest.flavor$
7 **return** $V.flavor$

[1] www.aws.amazon.com/autoscaling
[2] www.cloud.google.com/compute/docs/autoscaler
[3] www.aws.amazon.com/solutions/implementations/ops-automator
[4] www.azure.microsoft.com/services/automation
[5] www.aws.amazon.com/aws-cost-management/aws-cost-optimization/right-sizing
[6] www.cloudability.com
[7] www.zabbix.com
[8] www.prometheus.io
[9] www.aws.amazon.com/cloudwatch

3.1 Filtering step

To select a flavor for a VM, it is important to take into account each resource to ensure that the performance of the application running on the VM is not degraded. Our hypothesis here is that the performance is degraded if the resource capacity is lower than its utilization. For this reason, a filtering is performed to remove from the candidate list all flavors that do not offer the minimum amount of resources needed. For example, if an application is using 8 vCPUs and 25.6GiB, the filtering step would remove all the flavors that have less than this amount of vCPUs and memory size.

The algorithm analyzes the utilization in a time frame, not predicting the future needs nor checking its past history. This approach may recommend flavors that do not match the future demands of the application, in case the initial samples indicate that the application needs only a fraction of the amount of memory, for example. Without predicting the application's needs, another way to get this information is through user annotations, which tells the algorithm what are the minimum capacities of each resource. However, this conflict with our premise of new cloud users. With these annotations, if an application demands different amounts of resources over time, the algorithm avoids selecting sub-optimal flavors or ones that may degrade the application's performance.

3.2 Searching step

After the filtering step, the algorithm has a list of candidates that have the minimum amount of resources required by the application. The next step is to search for the optimal VM flavor on this list. Our goal is to select a VM flavor that keeps the current performance and reduces the execution cost.

As our study case focuses on parallel workloads, we use the vCPU utilization rate to search for the optimal flavor. The research question we try to answer is: *given an application that is executing in a VM of flavor A, is it possible to recommend a flavor B that keeps the application's performance and reduces its execution cost by knowing only the application's vCPU utilization rate in flavor A?*

3.2.1 vCPU-heuristic. Considering this, our hypothesis is: *if the current flavor has Z vCPUs and the application vCPU utilization rate is X%, $0 \leq X \leq 100$, then, the application would have the same (or better) performance on another flavor with at least $\frac{X \times Z}{100}$ vCPUs.*

Based on this hypothesis, there is the **vCPU-heuristic**, which defines a search space as the list of the flavors with a number of vCPUs that is equal to the *effective use of vCPUs*, which is the number of vCPUs in the current flavor multiplied by its utilization rate. The cheapest flavor in this list is the selected flavor, but only if it is cheaper than the current one. In other words, if only 50% of 16 vCPUs in the current VM are being utilized, the search space is composed of all VM flavors with 8 vCPUs. If our hypothesis is correct, then the performance will not be reduced and the execution cost will be reduced on the new flavor.

The effective use of vCPUs is calculated in line 2 of Algorithm 1. Then, the getSearchList() function is called in line 3 (green line) and returns the vCPU amount available that satisfies the effective use. More about this function will be discussed later. Once the vCPU amount of the search space is defined, the getCheapest() function returns the cheapest flavor with this amount. If the cheapest flavor is cheaper than the current, it is returned. Otherwise, the current flavor is considered the optimal one and returned.

Table 1 shows experimental results in which we executed the LU application from NPB[10] using 4 threads in different flavors from AWS. Next section provides more information about the experiment setup and the applications. In this table, the *relative performance* is calculated for each flavor in relation to the execution in the `r5.2xlarge` flavor (the one that presented the best performance).

Table 1: Previous results of LU application limited by 4 threads

Flavor	#vCPUs	Price (USD/hour)	vCPU utilization	Relative performance	Cost (USD)
`r5a.x`	4	$0.23	100%	1.97	$0.0011
`r5.x`	4	$0.25	100%	1.58	$0.0009
`t3.2x`	8	$0.33	50%	1.06	$0.0008
`r5a.2x`	8	$0.45	50%	1.43	$0.0015
`r5.2x`	8	$0.50	50%	1.00	$0.0012
`i3.2x`	8	$0.62	50%	1.17	$0.0017

As the application is using four threads, the heuristic would select the cheapest flavor with 4 vCPUs, the `r5a.x`. However, this flavor presented a relative performance of 1.97. In fact, the flavors with the exact amount of vCPUs that are utilized degrades more the performance. The other VM flavors, with twice the amount of vCPU that are utilized by the application, show smaller performance differences in comparison to the best one.

In IaaS, the CPU resource is provisioned in terms of virtual CPUs (vCPUs), which are usually mapped to hardware threads. In this context, in case the physical core contains Simultaneous Multi Threading, multiple vCPUs are mapped to a single physical core. As a consequence, on processors equipped with the *hyper-threading* technology, which have two hardware threads per core, two vCPUs are mapped to each core. This mapping explains why using 100% of vCPUs on a flavor with 4 vCPUs does yield the same performance of using 50% of vCPUs on a flavor with 8 vCPUs. In the first case, the application is using four hardware threads (vCPUs) that are mapped to two physical cores, causing the threads to compete for core resources. In the second case, the application is using four vCPUs that are likely to be mapped to four physical cores, allowing each thread to have exclusive access to its own core.

3.2.2 core-heuristic. Based on the previous observation, we propose a second hypothesis: *the performance of an application that has a vCPU utilization rate of X%, $0 \leq X \leq 100$, in a flavor with Z vCPUs is lower or equal to the performance of the same application in a flavor with $\frac{2 \times X \times Z}{100}$ vCPUs.*

This is the principle for our second heuristic, that is called **core-heuristic**. This heuristic defines the search space as the list of the flavors with twice the amount of the current effective use of vCPUs, *i.e.*, the current vCPU utilization rate multiplied by the amount of vCPUs on the current flavor. As in the previous heuristic, the cheapest flavor in this set is identified and selected, unless it is not cheaper than the current one. This heuristic is also expressed by Algorithm 1, however, using the red line instead of the green one.

[10] www.nas.nasa.gov/publications/npb.html

The only difference is that $effective$ is multiplied by 2; in this way, the getCheapest() function returns the cheapest flavor that have at least twice the effective use of vCPUs on the current flavor. Notice that this heuristic can be easily adapted for SMT-based systems with more than 2 hardware threads per core.

3.2.3 Price-per-vCPU heuristics. So far, the getCheapest() function selects the cheapest flavor using the *VM price* metric. However, we can also use the *price per vCPU* metric to make this choice. Therefore, two extra heuristics are proposed: **vCPU-ppv-heuristic** and **core-ppv-heuristic**.

3.3 vCPU amount available

It is possible to happen situations in which there are no flavors with the same amount of vCPUs as the effective use in the *candidates* list received by the getSearchList() function. This can happen due to a lack of flavors offered by the provider or due to the filtering step. Considering this, the function returns the vCPU amount available with at least the parameter received ($effective$ for vCPU-heuristic and $2 \times effective$ for core-heuristic). If this parameter is greater than the maximum available, the maximum is returned.

4 EXPERIMENTAL RESULTS

4.1 Experiment design

We evaluated the heuristics using applications from the NPB 3.4v[11] benchmark, which rely on threads to accelerate their execution with multiple processing cores. Table 2 summarizes the applications used in our experiments, the restrictions on the number of threads, and the maximum amount of memory required. These applications have different behaviors and resource requirements. Some of them focus on intensive computations, others on memory randomness or access, and others on irregular communications.

Table 2: Applications used in the experiments

Applications	Input	#threads	Maximum memory usage
BT	D	square number	≈ 25GB
SP	D	square number	≈ 22GB
CG	D	power-of-two number	≈ 20GB
EP	E	power-of-two number	≈ 1GB
LU	D	power-of-two number	≈ 12GB
FT	D	power-of-two number	≈ 115GB
IS	D	power-of-two number	≈ 25GB

Table 3 shows the Amazon Web Services (AWS) flavors used. The experiments were conducted with the Canonical, Ubuntu, 18.04 LTS, amd64 AMI (ami-07c1207a9d40bc3bd).

4.1.1 Paramount Interaction. The execution time of these applications can become unfeasible for our financial constraints. For this reason, we used the *paramount iteration* technique to estimate the relative performance of flavors [5]. The methodology consists of using the execution time of the first *paramount iterations* as a proxy to the performance of the application on each flavor. With this technique, it is possible to infer the performance relation between the different flavors by limiting the execution time of the application in each flavor. We used the performance of the first 40 *paramount iterations* as a proxy for the applications' performance.

[11] www.nas.nasa.gov/publications/npb.html

Table 3: VM flavors in AWS

Flavor	#vCPUs	#CPUs	Mem (GiB)	Price[a] (USD/hour)	Price/vCPU
r5a.x	4	2	32	$0.23	$0.057
r5.x	4	2	32	$0.25	$0.063
t3.2x	8	4	32	$0.33	$0.042
r5a.2x	8	4	64	$0.45	$0.057
r5.2x	8	4	64	$0.50	$0.063
i3.2x	8	4	61	$0.62	$0.078
c5a.4x	16	8	32	$0.62	$0.039
c5.4x	16	8	32	$0.68	$0.043
c5n.4x	16	8	42	$0.86	$0.054
r5.4x	16	8	128	$1.01	$0.063
i3.4x	16	8	122	$1.25	$0.078
r5a.8x	32	16	256	$0.90	$0.028
c5a.8x	32	16	64	$1.23	$0.039
r5.8x	32	16	256	$2.02	$0.063
r5n.8x	32	16	256	$2.38	$0.075
c5.9x	36	18	72	$1.53	$0.043
c5n.9x	36	18	96	$1.94	$0.054
c5.12x	48	24	96	$2.04	$0.043
r5.12x	48	24	384	$3.02	$0.063
r5n.12x	48	24	384	$3.58	$0.075

[a] On-demand pricing for Linux instances in US East (Ohio) region, gathered on 12/15/2020, on: aws.amazon.com/ec2/pricing/on-demand/.

4.1.2 Evaluate Approach. We evaluated the heuristics in two scenarios: `Few-Threads` and `Max-Threads`. In the `Few-Threads` scenario, the number of threads is limited to induce severe resource underutilization. In this case, most applications are executed with four threads, except FT, which is also executed with sixteen threads. In the `Max-Threads` scenario, we maximize the number of threads on the flavor without overloading it. In this case, we limit the number of threads to the maximum value possible that is smaller than the number of vCPUs available in the flavor. For example, BT, which requires a *square number* of threads, is executed with 4 (2^2) threads in flavors with 4 and 8 vCPUs, and with 36 (6^2) threads in flavors with 36 and 48 vCPUs. While LU, which requires a *power-of-two number* of threads, is executed with 4 (2^2) threads in flavors with 4 vCPUs, with 8 (2^3) threads in flavors with 8 vCPUs, and with 32 (2^5) in flavors with 32, 36 and 48 vCPUs. In this scenario there is less underutilization and more variation of search spaces for the heuristics as there are executions with different number of threads. All experiments were executed 3 times in different days. We analyzed the results and verified that the performance was consistent between days, which indicates that there was little or no performance variations caused by sharing the physical machines with other cloud users.

The evaluation is based on *performance speedup* and *cost reduction rate* of the selected flavor in relation to the current flavor. In this way, we can analyze the heuristics' quality by verifying if they

are reducing the cost and maintaining the performance. The performance is measured based on execution time in seconds while the cost is calculated by multiplying the execution time by the flavor price, in USD per second. It is important to state that the calculation uses the execution with the same number of threads in both selected and current flavors. For example, if a heuristic selects `t3.2xlarge` for the execution with 4 threads in `r5.xlarge`, the cost reduction rate and the performance speedup are calculated using the execution with 4 threads in both flavors. All experiments' scripts, the applications instrumented to report the *paramount iterations* and results are available at GitHub [12].

4.2 Heuristics' selections

Table 4 shows the selections of each heuristic for the LU application in the `Few-Threads` scenario. In the third row, when executing the application with four threads on the `t3.2xlarge` flavor, the vCPU-heuristic selects the `r5a.xlarge`, while the vCPU-ppv-, core- and core-ppv-heuristics preserves the same flavor.

Table 4: Selection of each heuristic for the execution of LU application in the `Few-Threads` scenario

Current VM				VM flavor selected by each heuristic			
flavor	Price	Price /vCPU	# thread	vCPU-	vCPU-ppv-	core-	CPU-ppv-
r5a.x	0.23	0.058	4	r5a.x	r5a.x	r5a.x	t3.2x
r5.x	0.25	0.062	4	r5a.x	r5a.x	r5.x	t3.2x
t3.2x	0.33	0.041	4	r5a.x	t3.2x	t3.2x	t3.2x
r5a.2x	0.45	0.056	4	r5a.x	r5a.2x	t3.2x	t3.2x
r5.2x	0.5	0.062	4	r5a.x	r5a.x	t3.2x	t3.2x
i3.2x	0.62	0.077	4	r5a.x	r5a.x	t3.2x	t3.2x
c5a.4x	0.62	0.039	4	r5a.x	c5a.4x	t3.2x	c5a.4x
c5.4x	0.68	0.043	4	r5a.x	c5.4x	t3.2x	t3.2x
c5n.4x	0.86	0.054	4	r5a.x	c5n.4x	t3.2x	t3.2x
r5.4x	1.01	0.063	4	r5a.x	r5a.x	t3.2x	t3.2x
i3.4x	1.25	0.078	4	r5a.x	r5a.x	t3.2x	t3.2x
r5a.8x	0.9	0.028	4	r5a.x	r5a.8x	t3.2x	r5a.8x
c5a.8x	1.23	0.038	4	r5a.x	c5a.8x	t3.2x	c5a.8x
r5.8x	2.02	0.063	4	r5a.x	r5a.x	t3.2x	t3.2x
r5n.8x	2.38	0.074	4	r5a.x	r5a.x	t3.2x	t3.2x
c5.9x	1.53	0.043	4	r5a.x	c5.9x	t3.2x	t3.2x
c5n.9x	1.94	0.054	4	r5a.x	c5n.9x	t3.2x	t3.2x
c5.12x	2.04	0.043	4	r5a.x	c5.12x	t3.2x	t3.2x
r5.12x	3.02	0.063	4	r5a.x	r5a.x	t3.2x	t3.2x
r5n.12x	3.58	0.075	4	r5a.x	r5a.x	t3.2x	t3.2x

The number of vCPUs in effective use in this experiment is always four. Therefore, the vCPU-heuristic selects the cheapest flavor with 4 vCPUs (`r5a.x`) for all executions. The vCPU-ppv-heuristic uses the same search space, however, there are flavors with more capacity that are cheaper than the `r5a.x` flavor when considering the *price per vCPU* metric. When the current flavor is cheaper than the cheapest in the search space, the heuristic does not change the flavor. For this reason, this heuristic opts to keep the flavor even when there is high underutilization.

The core-heuristic, on the other hand, selects the cheapest flavor with 8 vCPUs (`t3.2x`) for all executions; except for the scenarios in the first two rows, because the current flavors are cheaper than `t3.2x`. As vCPU-ppv-, the core-ppv-heuristic avoids changing the current flavor because it is cheaper using the *price per vCPU* metric.

As all applications use four threads and have the same candidate flavors, the heuristics present the same selections for them. Except for FT, that has high memory usage resulting in less candidate flavors after the filtering step.

On the `Max-Threads` scenario, the applications use different amounts of threads, which lead to different search spaces and a larger variety of selections. Table 5 shows the heuristics' selections for the LU application. The vCPU-heuristic always selects the cheapest flavor with the vCPU amount equal to the number of threads. The vCPU-ppv-heuristic showed same results, because the cheapest flavor in each group of vCPU amount is the same using *VM price* and *price per vCPU* metrics. The core-heuristic tends to maintain the current flavor, because the cheapest with the double amount of the effective use of vCPUs tends to be more expensive, as they have more resources. The core-ppv-heuristic, on the other hand, tends to change the flavor, as it uses *price per vCPU* metric.

Table 5: Selection of each heuristic for the execution of LU application in the `Max-Threads` scenario

Current VM				VM flavor selected by each heuristic			
flavor	Price	Price /vCPU	# thread	vCPU-	vCPU-ppv-	core-	CPU-ppv-
r5a.x	0.23	0.058	4	r5a.x	r5a.x	r5a.x	t3.2x
r5.x	0.25	0.062	4	r5a.x	r5a.x	r5.x	t3.2x
t3.2x	0.33	0.041	8	t3.2x	t3.2x	t3.2x	c5a.4x
r5a.2x	0.45	0.056	8	t3.2x	t3.2x	r5a.2x	c5a.4x
r5.2x	0.5	0.062	8	t3.2x	t3.2x	r5.2x	c5a.4x
i3.2x	0.62	0.077	8	t3.2x	t3.2x	i3.2x	c5a.4x
c5a.4x	0.62	0.039	16	c5a.4x	c5a.4x	c5a.4x	r5a.8x
c5.4x	0.68	0.043	16	c5a.4x	c5a.4x	c5.4x	r5a.8x
c5n.4x	0.86	0.054	16	c5a.4x	c5a.4x	c5n.4x	r5a.8x
r5.4x	1.01	0.063	16	c5a.4x	c5a.4x	r5a.8x	r5a.8x
i3.4x	1.25	0.078	16	c5a.4x	c5a.4x	r5a.8x	r5a.8x
r5a.8x	0.9	0.028	32	r5a.8x	r5a.8x	r5a.8x	r5a.8x
c5a.8x	1.23	0.038	32	r5a.8x	r5a.8x	c5a.8x	c5a.8x
r5.8x	2.02	0.063	32	r5a.8x	r5a.8x	r5.8x	c5.12x
r5n.8x	2.38	0.074	32	r5a.8x	r5a.8x	c5.12x	c5.12x
c5.9x	1.53	0.043	32	r5a.8x	r5a.8x	c5.9x	c5.9x
c5n.9x	1.94	0.054	32	r5a.8x	r5a.8x	c5n.9x	c5.12x
c5.12x	2.04	0.043	32	r5a.8x	r5a.8x	c5.12x	c5.12x
r5.12x	3.02	0.063	32	r5a.8x	r5a.8x	c5.12x	c5.12x
r5n.12x	3.58	0.075	32	r5a.8x	r5a.8x	c5.12x	c5.12x

As our evaluation is focused on vCPU utilization, the selections do not make use of any other aspect of the application. Due to this, these selections are the same for the applications that uses power-of-two number of threads. Except for FT, which has different candidate flavors due to the larger amount of memory usage. The applications

[12] www.github.com/lmcad-unicamp/hpcc-monitor/tree/master/experiments

with square number of threads present other selections, because they have different amount of threads in use in each flavor. However, the heuristics present similar tendencies in these applications.

Nevertheless, these heuristics have different impact in each application. Figure 1 presents the performance speedup and relative cost reduction of the heuristics in each application on the `Few-Threads` scenario. The horizontal line represent the performance speedup 1.00, so, if the geometric mean is upper this line, the performance is improved, if it is lower, the performance is degraded. The vertical line is cost reduction at 1.00 so, points to the left of this line represent cost reduction, while, to the right, cost increase. Notice that some of the heuristics tend to degrade the performance less than others, sometimes even improving it. When it comes to the cost, it is always reduced, but the relation between the heuristics is more mixed. The core-heuristic and the core-ppv-heuristic reduce more the cost than the other two heuristics. However, in EP application, the vCPU-ppv-heuristic is the one that reduces the most.

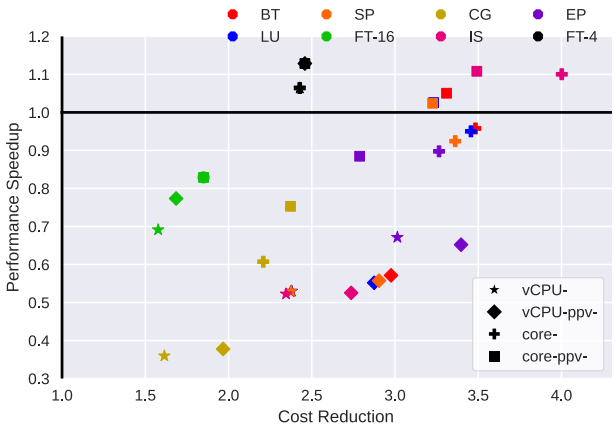

Figure 1: Performance speedup and relative cost reduction of each heuristic in each application on the `Few-Threads` scenario

The cost reduction in the `Few-Threads` scenario is high, mainly because the heuristics suggest replacing expensive flavors (with lots of computing resources) by one of the cheapest (with few vCPUs and little memory) due to the high underutilization. For the `Max-Threads` scenario, the cost reduction is smaller, as there is not much room for reducing the amount of computing resources (*e.g.*, vCPUs) without affecting performance. Figure 2 shows how the vCPU utilization affects the cost reduction achieved by each heuristic on both scenarios. As expected, all heuristics have a tendency to achieve higher cost reductions as the utilization rate is smaller.

Figure 3 shows the performance speedup in relation to the vCPU utilization for experiments in both scenarios. The vCPU- and vCPU-ppv-heuristics tend to degrade the application's performance more when the utilization rate is low. This occurs because, when an application is using less than half of the vCPU capacity – having at most one thread per core –, these heuristics select flavors that increase the vCPU utilization, resulting in the scheduling of two threads per core – due to the hyper-threading. In other words, the processing core resources must now be shared between two threads.

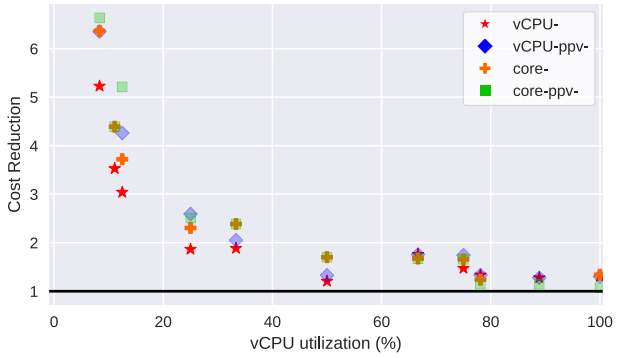

Figure 2: Cost reduction's geometric mean by the vCPU utilization rate of each heuristic

The same does not happen when the application is using more than the half of the vCPU capacity, because it is already using more than one thread per core and the heuristics would select a flavor with an amount of vCPUs similar to the current one. Different from the other situation, the amount of resources would not be cut by half.

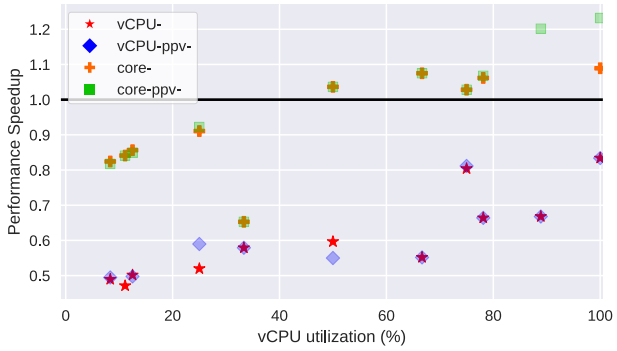

Figure 3: Performance speedup's geometric mean by the vCPU utilization rate of each heuristic

Another observation is that, with more than 50% of utilization, the core- and core-ppv-heuristics improve the performance. This happens because the vCPU capacity is doubled. The opposite happens when the utilization is less than 50%, the heuristics pose a small impact on the performance because they select a flavor with at least double the vCPU amount as the effective use, but capacity is still lowered and may be responsible for this small degradation (*e.g.*, if an application is using 8 vCPUs of a flavor with 32, the heuristics would select a flavor with 16).

4.3 Heuristics' overview

Figure 4 presents the geometric means for the performance speedups and cost reductions achieved by each heuristic in the `Few-Threads` scenario. Taking the point (1.00, 1.00) as the origin of our Cartesian coordinate system, we searched for a heuristic that would produce results in the upper right quadrant. Unfortunately, none of the geometric means are in the upper right quadrant. Nonetheless, the core- and core-ppv-heuristics were able to achieve high cost reductions with low performance degradation in this scenario.

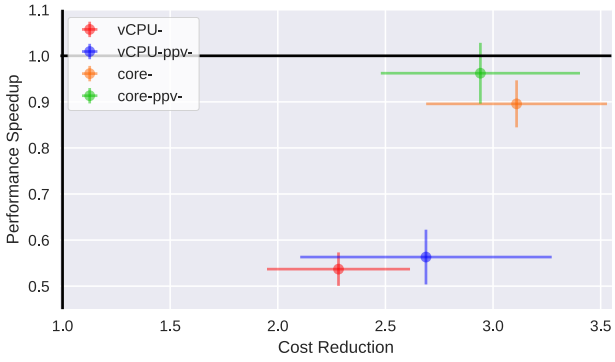

Figure 4: Performance speedup and cost reduction geometric's means for each heuristic on the `Few-Threads` scenario. The horizontal and vertical lines indicate the 95% confidence intervals

The geometric means and confidence intervals indicate the overall tendency, however, we are also interested in knowing the frequencies in which each heuristic produces results in each quadrant. Figure 5 shows the percentage of the experiments in each quadrant for the selections of each heuristic on the `Few-Threads` scenario. Only the core-ppv-heuristic presented selections at the upper left quadrant (performance improvement and cost increase), exactly one of the selections. The others showed selections at the lower left quadrant, in which the cost is increased and the performance degraded. For the vCPU-heuristic, these represent 9.7% of the selections, for vCPU-ppv-heuristic, 4.4%, and for core-heuristic, 1.5%. Most of the selections are in lower right quadrant, as expected. However, some selections increased the performance while reducing the cost (upper-right quadrant): the vCPU- and vCPU-ppv-heuristics have two selections in the upper right quadrant, while for the other two these are around 18% of the selections.

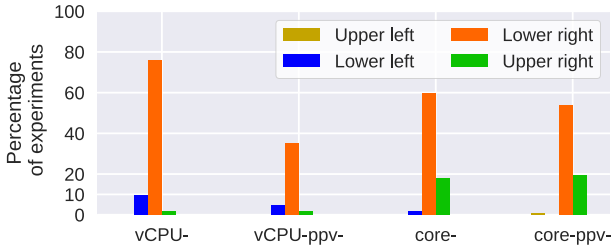

Figure 5: Percentage of the experiments in each quadrant for the selections of each heuristic on the `Few-Threads` scenario

Figure 6 presents the geometric means for the performance speedups and cost reductions achieved by each heuristic in the `Max-Threads` scenario. Here, the cost reduction is lower, but the performance is less degraded and often improved. The vCPU- and vCPU-ppv-heuristics achieve similar results, presenting between 20% and 30% of performance degradation. However, the core- and core-ppv-heuristics increase the performance while reducing the cost. In the last case, there is a trade-off: the core-heuristic achieves better cost reductions, while the core-ppv-heuristic presents higher performance improvements.

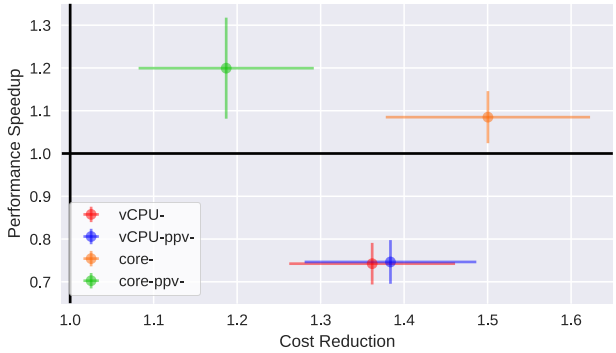

Figure 6: Performance speedup and cost reduction geometric's means for each heuristic on the `Max-Threads` scenario. The horizontal and vertical lines indicate the 95% confidence intervals

Figure 7 shows the percentage of the experiments in each quadrant for the selections of each heuristic on the `Max-Threads` scenario. The executions in which the selected flavor is the same as the current are considered in the proportion. The vCPU- and vCPU-ppv-heuristic have more than 40% of the selections in the lower right quadrant, for the other two, these are less than 3%. The selections of these other two heuristics are concentrated at the upper right quadrant: almost 20% for core-heuristic and more than 40% for core-ppv-heuristic. For the vCPU- and vCPU-ppv-heuristics, these are less than 10%. The worst cases – the ones in lower left quadrant – in core-heuristic are not frequent, being less than 3% of the selections. For the other heuristics, they are almost 10%. The vCPU-ppv-heuristic is the only one that present selections in the upper left quadrant, being 7.8% of the total.

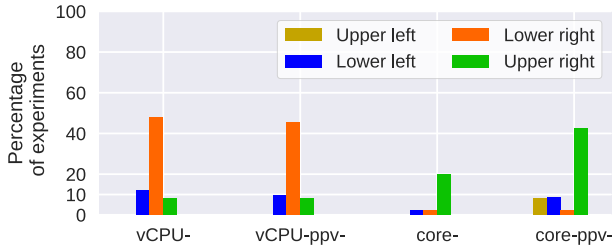

Figure 7: Percentage of the experiments in each quadrant for the selections of each heuristic on the `Max-Threads` scenario

4.4 Heuristic's conservatism

In the previous analysis, we were not considering the selections that are the same as the current flavor. We removed them because, when the selected flavor is the same as the current, there are no performance nor cost changes.

In general, changing the flavor of a VM requires stopping the application execution and restoring it. This, sometimes, can be

expensive due to the delay introduced by this process. Moreover, it may not be simple to restore the execution – needing to save checkpoints of the application state to avoid executing again what already has been executed, as re-executing from the beginning can be costly. In this context, it is better to keep the same flavor than replacing it by another one with similar costs. Hence, it is of our interest to evaluate two situations: the conservatism of heuristics – that is, how often the heuristic recommends changing the current flavor – and if the recommended flavors present similar costs.

To perform this analysis, we analyze the results on both scenarios and consider cost results to be similar when their differ by no more than 5% (1.00 USD and 0.96 USD are considered similar costs). Figure 8 (a) shows, for each heuristic, the amount of experiments they select a flavor equal or different from the current flavor. It is clear that vCPU-ppv- and core-heuristics are more conservative in recommending new flavors, while vCPU- and core-ppv-heuristics are the most aggressive. The conservatism of the heuristics comes from the fact that the algorithm returns the same flavor if the cheapest flavor in the search space is more expensive than the current one (line 5 of Algorithm 1). As the search space of the core-heuristic contains flavors with more capacity and, consequently, more expensive, it tends to be more conservative. The opposite happens with the vCPU-heuristic, in which the search space is composed mostly of flavors with less capacity, so, they are cheaper than the current one and tend to change it. The vCPU-ppv- and core-ppv-heuristics, on the other hand, twist these behaviors due to the use of the *price per vCPU* metric.

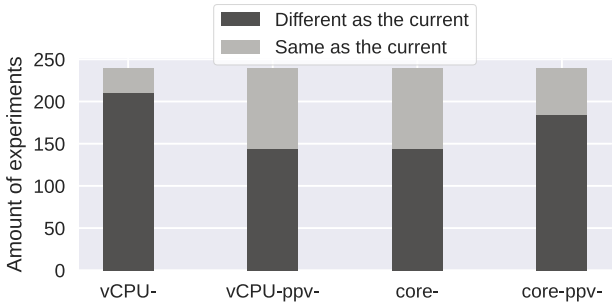

Figure 8: Flavor changing frequency for each heuristic in both scenarios

Figure 9 (b) shows that the selections that are different from the current flavor but have similar costs are not very common to happen, less than 4.6% of the total selections in the vCPU- and vCPU-ppv-heuristics, 2% in core-ppv-heuristic and only one occurrence in the core-heuristic. In general, the heuristics tend to achieve different costs and performances when changing the flavor.

The main criteria that causes the heuristics to be conservative in Algorithm 1 is the *if statement* in line 5. Hence, we evaluated the impact of this criteria on the cost and performance of the selections by removing this conditional statement. Note that, by removing this statement, the selections of vCPU-ppv-heuristic become equal to the ones of vCPU-heuristic, and the selections of core-ppv-heuristic equal to core-heuristic. This happens because the cheapest flavor in each subset of flavors with the same amount of vCPUs is the

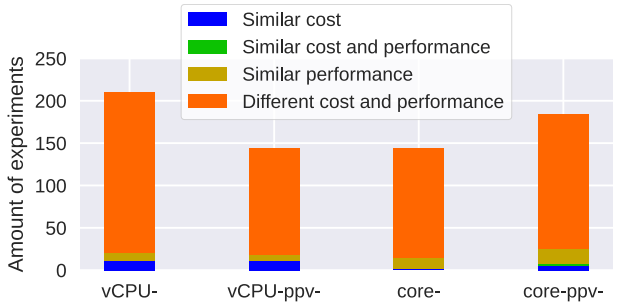

Figure 9: Selection performance and cost in relation to the current for each heuristic for experiments in both scenarios

same using the *VM price* or *price per vCPU* metrics, as indicated in Table 3. So, if this criteria is removed, the heuristics will always select the cheapest flavor in the search space using the price per vCPU, which also happens when using the VM price criteria.

Figure 10 presents the costs and performances geometric means for the selections from the version without the conservative criteria when the selections are different when compared with the version with the criteria. Therefore, this figure shows the impact of the criteria. The vCPU-heuristic has only two occurrences, therefore removing the conservative criteria causes the vCPU-heuristic to recommend flavors that offer worst performance and higher cost. The vCPU-ppv-heuristic reduces the cost, but with performance degradation. The core-heuristic improves the performance and increases a little the cost, while core-ppv-heuristic tends to reduce the cost while improving the performance (with high confidence interval). This shows us that the conservative criteria for core-ppv-heuristic is good (with high variation), while for others are not.

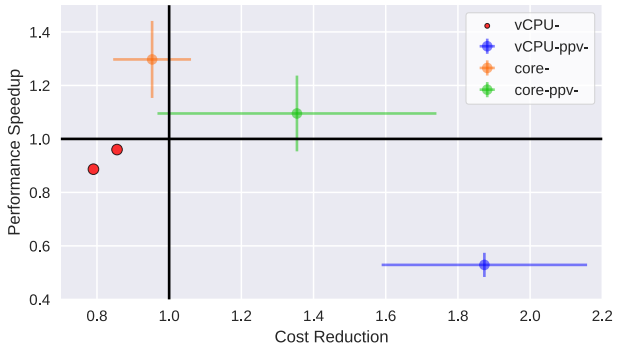

Figure 10: Performance speedup and cost reduction's geometric means for each heuristics without conservative criteria for the selections that are different when compared with the version with the criteria. The horizontal and vertical lines indicate the 95% confidence intervals

5 CONCLUSIONS & FUTURE WORKS

The main objective of this work is to develop a methodology to optimize the cost of parallel workloads in cloud computing for new users, by recommending flavors that would reduce the cost based

solely on resource's utilization data in the current one. The constraint is to avoid application's performance loss. We proposed four heuristics that, based on the application vCPU utilization rate on the current flavor, aim at identifying alternative flavors that reduce the cost without compromising the applications' performance. We evaluated them with seven parallel applications on 20 AWS EC2 flavors, a total of 239 different experiments.

The vCPU- and vCPU-ppv-heuristics presented good cost reductions with high performance degradation. Our analysis suggests that this degradation comes from the fact that vCPUs are mapped to hardware threads, instead of computing cores. In the case of hyper-threading, each computing core is shared by two hardware threads. The core-heuristic overcomes this issue, presenting the best cost reductions and tending to maintain the performance. It was capable of reducing the cost, on average, by 1.5× (3.0×) on high (low) vCPU-utilization rate scenarios. Also, it presented the lowest frequency of selections that increased the cost or degraded the performance – less than 5% of the selections. Even though this heuristic gives the best results, the core-ppv-heuristic may be more suitable when the context is more sensitive to loosing performance.

Even though the heuristics were evaluated only with multi-threaded applications, we expect these heuristics to work on multi-process applications on a variety of scenarios. For CPU-bound applications, recommending replacing a subset of the VMs could be useful to reduce underutilization caused by applications that suffer from load-balancing issues. It also makes sense to adjust the infrastructure in cases where the processes use different amounts of memory. For network-bound applications, applying these heuristics to optimize the VMs individually may not be directly applicable, as the communication performance may depend on factors that are outside the scope of the VMs, such as the VM location on the datacenter, or the communication switches load. As future work, they should be evaluated with multi-process applications and could be extended to work with network-bound.

ACKNOWLEDGMENTS

The authors thank Petrobras, CNPq (314645/2020-9), FAPESP (CEPID 2013/08293-7), and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) for their financial support. The authors also thank the Multidisciplinary High Performance Computing Laboratory (LMCAD) of the Institute of Computing of Unicamp for the computational support.

REFERENCES

[1] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation*. USENIX.

[2] Shivnath Babu. 2010. Towards Automatic Optimization of MapReduce Programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. Association for Computing Machinery.

[3] Matt Baughman, Ryan Chard, Logan Ward, Jason Pitt, Kyle Chard, and Ian Foster. 2018. Profiling and Predicting Application Performance on the Cloud. In *11th International Conference on Utility and Cloud Computing*.

[4] Muhammad Bilal, Marco Canini, and Rodrigo Rodrigues. 2020. Finding the Right Cloud Configuration for Analytics Clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing*.

[5] Jeferson R. Brunetta and Edson Borin. 2019. Selecting Efficient Cloud Resources for HPC Workloads. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*.

[6] Eun-Kyu Byun, Yang-Suk Kee, Jin-Soo Kim, and Seungryoul Maeng. 2011. Cost optimized provisioning of elastic resources for application workflows. *Future Generation Computer Systems* 27 (2011).

[7] Jay Chapel. 2019. Cloud Waste To Hit Over $14 Billion in 2019. https://devops.com/cloud-waste-to-hit-over-14-billion-in-2019/.

[8] Javier Espadas, Arturo Molina, Guillermo Jimenez, José Molina Espinosa, Raul Ramirez-Velarde, and David Concha. 2013. A tenant-based resource allocation model for scaling Software-as-a-Service applications over Cloud computing infrastructures. *Future Generation Computer Systems* 29 (2013).

[9] Ayat Fekry, Lucian Carata, Thomas F. J.-M. Pasquier, Andrew Rice, and Andy Hopper. 2020. Tuneful: An Online Significance-Aware Configuration Tuner for Big Data Analytics. *CoRR* abs/2001.08002 (2020).

[10] G. Galante and L. C. E. de Bona. 2012. A Survey on Cloud Computing Elasticity. In *2012 IEEE Fifth International Conference on Utility and Cloud Computing*.

[11] Chin-Jung Hsu, Vivek Nair, Vincent W. Freeh, and Tim Menzies. 2018. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In *38th International Conference on Distributed Computing Systems*.

[12] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent Freeh. 2018. Micky: A Cheaper Alternative for Selecting Cloud Instances. In *2018 IEEE 11th International Conference on Cloud Computing*.

[13] E. Hwang and K. H. Kim. 2012. Minimizing Cost of Virtual Machines for Deadline-Constrained MapReduce Applications in the Cloud. In *2012 ACM/IEEE 13th International Conference on Grid Computing*.

[14] Gueyoung Jung, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and Calton Pu. 2009. A Cost-Sensitive Adaptation Engine for Server Consolidation of Multitier Applications. In *Middleware 2009*.

[15] Stefan Kehrer and Wolfgang Blochinger. 2019. A survey on cloud migration strategies for high performance computing. In *Proceedings of the 13th Symposium and Summer School on Service-Oriented Computing*.

[16] Derdus Kenga, Vincent Omwenga, and Patrick Ogao. 2019. Virtual Machine Sizing in Virtualized Public Cloud Data Centres. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* (2019).

[17] Woongsup Kim and Jaha Mvulla. 2013. Reducing Resource Over-Provisioning Using Workload Shaping for Energy Efficient Cloud Computing. *Applied Mathematics and Information Sciences* 7 (2013).

[18] Wesley Lloyd, Shrideep Pallickara, Olaf David, Mazdak Arabi, Tyler Wible, and Jeffrey Ditty. 2015. Demystifying the Clouds: Harnessing Resource Utilization Models for Cost Effective Infrastructure Alternatives. *IEEE Transactions on Cloud Computing* (2015).

[19] Marco A. S. Netto, Rodrigo N. Calheiros, Eduardo R. Rodrigues, Renato L. F. Cunha, and Rajkumar Buyya. 2018. HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. *Comput. Surveys* 51 (2018).

[20] S. Pandey, L. Wu, S. M. Guru, and R. Buyya. 2010. A Particle Swarm Optimization-Based Heuristic for Scheduling Workflow Applications in Cloud Computing Environments. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*.

[21] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*.

[22] Mina Sedaghat, Francisco Hernandez-Rodriguez, and Erik Elmroth. 2013. A Virtual Machine Re-Packing Approach to the Horizontal vs. Vertical Elasticity Trade-off for Cloud Autoscaling. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*.

[23] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. 2011. A Cost-Aware Elasticity Provisioning System for the Cloud. In *31st International Conference on Distributed Computing Systems*.

[24] Thamarai Selvi Somasundaram and Govindarajan Kannan. 2014. CLOUDRB: A framework for scheduling and managing High-Performance Computing applications in science cloud. *Future Generation Computer Systems* 34 (2014).

[25] Sen Su, Jian Li, Qingjia Huang, Xiao Huang, Kai Shuang, and Jie Wang. 2013. Cost-efficient task scheduling for executing large programs in the cloud. *Parallel Comput.* 39 (2013).

[26] William F. C. Tavares, Marcio R. Miranda, and Edson Borin. 2021. Quantifying and Detecting HPC Resource Wastage in Cloud Environments. In *Proceedings of the 2021 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*.

[27] Luis Tomás and Johan Tordsson. 2013. Improving Cloud Infrastructure Utilization through Overbooking. In *ACM Cloud and Autonomic Computing Conference*.

[28] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation*.

[29] Ai Xiao, ZhiHui Lu, Junnan Li, Jie Wu, and Patrick C.K. Hung. 2019. SARA: Stably and quickly find optimal cloud configurations for heterogeneous big data workloads. *Applied Soft Computing* 85 (2019).

[30] N. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton J. Smith, and R. Katz. 2017. Selecting the best VM across multiple public clouds: a data-driven performance modeling approach. *Symposium on Cloud Computing* (2017).