Optimizing Multi-Objective Evolutionary Algorithms to Enable Quality-Aware Software Provisioning

Donia El Kateb[1,2], François Fouquet[1], Johann Bourcier[3], Yves Le Traon[1,2]
[1]Security, Reliability and Trust
Interdisciplinary Research Center, SnT, University of Luxembourg, Luxembourg
[2]Laboratory of Advanced Software SYstems (LASSY), University of Luxembourg, Luxembourg
[3]University of Rennes 1, IRISA, INRIA, Rennes, France
{donia.elkateb, francois.fouquet, yves.letraon}@uni.lu
{johann.bourcier}@inria.fr

***Abstract*—Elasticity is a key feature for cloud infrastructures to continuously align allocated computational resources to evolving hosted software needs. This is often achieved by relaxing quality criteria, for instance security or privacy because quality criteria are often conflicting with performance. As an example, software replication could improve scalability and uptime while decreasing privacy by creating more potential leakage points. The conciliation of these conflicting objectives has to be achieved by exhibiting trade-offs. Multi-Objective Evolutionary Algorithms (MOEAs) have shown to be suitable candidates to find these trade-offs and have been even applied for cloud architecture optimizations. Still though, their runtime efficiency limits the widespread adoption of such algorithms in cloud engines, and thus the consideration of quality criteria in clouds. Indeed MOEAs produce many dead-born solutions because of the Darwinian inspired natural selection, which results in a resources wastage. To tackle MOEAs efficiency issues, we apply a process similar to modern biology. We choose specific artificial mutations by anticipating the optimization effect on the solutions instead of relying on the randomness of natural selection. This paper introduces the *Sputnik* algorithm, which leverages the past history of actions to enhance optimization processes such as cloud elasticity engines. We integrate *Sputnik* in a cloud elasticity engine, dealing with performance and quality criteria, and demonstrate significant performance improvement, meeting the runtime requirements of cloud optimization.**

***Keywords*—MOEAs, Hyper-heuristics, Optimization, Cloud, Software Deployment.**

I. Introduction

a) ***Software Deployment in the Cloud is a Multi-Objective Optimization Problem****:* Cloud Computing paradigm leverages hardware and software resources to offer hosting capabilities that enable customers to get rid of the burden of maintaining their own applications by paying fees that vary on the CPU power consumption per hour or storage usage instead of paying for software license. When hosting a software in the cloud, cloud providers aim at both achieving the quality of service requirements that are stated in the Service Level Agreements (SLA) [30] and at reducing energy consumption. In several cases, these objectives are conflicting: An illustrating example is the one related to Virtual Machines (VMs) allocation: Cloud customers tend to isolate their workloads in separate VMs for security purposes while cloud providers aim at reducing the number of alive VMs to reduce energy consumption in the data-center. The resolution of these objectives has to be performed within acceptable time frame to cope with elasticity features such as resources dynamic auto-scaling to meet users needs.

b) ***MOEAs for Cloud Multi-Objective Optimization Problems Resolution****:* In [18], the authors have motivated the use of Search Based approaches for the resolution of cloud engineering problems. Multi-Objective Evolutionary Algorithms (MOEAs) [12], [36] is a class of search based approaches that addresses problems in which a decision maker aims at finding a solution that optimizes several conflicting objectives. MOEAs simulate population evolution to produce solutions exhibiting trade-offs between conflicting objectives such as grid jobs scheduler [16] as they are able to automate a set of configurations exploration [10]. A cloud infrastructure, characterized by its dynamic entities, is a typical example of self-adaptive system. MOEAs offer generic and reusable domain exploration capabilities which make them suitable candidate for self-adaptive systems. Self-Adaptive systems require run-time corrective actions [25], that will be made after the identification/selection of the best fitted configuration. A cloud infrastructure can be abstracted by a set of software resources that run on top of Virtual Machines (VMs) dynamically starting/stopping in physical machines. MOEAs are thus used nowadays in several design case studies [16], [24], such as self-adaptive cloud scheduling problems, to maintain conflicting quality characteristics [12], [36] such as system performance, cost and safety. Beyond their applicability for cloud optimizers, MOEAs offer the following advantages to set-up autonomous self-adaptive engines working "at run-time": (i) no need for predefined solutions, (ii) the incremental optimization process can be stopped on-demand, (iii) operating multi-objective optimization and finding trade-offs.

c) ***MOEAs Tuning is needed to make them appropriate for self-adaptive systems such as a cloud infrastructure****:* MOEAs are not the standard solution for designing a run-time adaptation engine. A major factor which threatens their adoption for self-adaptive systems such as a cloud infrastructure is the run-time resources consumption, which often requires ad-

hoc empirical tuning of such algorithms to meet performance needs. Indeed, as in the Darwinian theory [11], [32], the evolution process of MOEAs relies on random mutations to ensure proper domain exploration. This randomness leads to sub-optimal performance due to the creation of many dead-born evolution branches. Coello *et al* [10] report that even if MOEAs usage simplifies the design of automatic configuration engines, empirical fine-tuning of evolutionary search parameters is still needed to save computational costs. These results motivate the need for software engineering techniques to avoid MOEAs ad-hoc tuning and to provide reusable techniques and frameworks for run-time usage.

d) ***Besides, modern genetics is evolving****:* Nowadays, modern genetics does not only rely on natural evolution process. Instead, based on the founding work of Muller *et al* [31], artificial mutation is now widely used to save time and generation cycles for instance to produce genetically modified organisms (GMO) [8]. Instead of relying only on crossover and natural selection, Muller *et al* [31] studied artificial mutation using X-Ray to modify a fruit with an anticipated intent. These principles have led the genetic field to build instruments for such selective artificial mutations: the evolution process is accelerated by selecting some specific mutations that contribute to enhance a certain objective.

e) ***Our proposal to accelerate MOEAs****:* Going along the same line, we study how such principles could be adapted to MOEAs to accelerate the convergence by guiding the evolutionary algorithms through dynamically selected mutation operators. Our intuition is that operators applied in a smart and artificial way would provide better results than operators applied randomly, and in particular would reduce the number of useless solutions. Thus, the new algorithm we propose is no longer inspired by Darwinian evolution, but by "artificial mutation", based on a smart and dynamic selection of the best mutation operator to apply at a given step. By applying such operators in priority, we aim at orienting the evolution process of a given population in the right direction for the problem to solve.

In this paper, we present a hyper-heuristic [7], called *Sputnik*, inspired by artificial mutation. Our algorithm takes its name from a virus family which evolves and mutates together with their host in order to perfectly fit their environment and to replicate more quickly. In the same manner, *Sputnik* algorithm leverages a continuous ranking of operators according to their impact on fitness functions to smartly select dynamically the most relevant mutation operator as the search evolves.

We focus on performance as a key factor for run-time usage to reach faster acceptable trade-offs while saving computation time and generation cycles. For instance, the acceleration *Sputnik* provides is useful for adaptive systems when a solution/reaction has to be found in a short time. We evaluate our approach on a cloud reasoning engine that is able to continuously provision customers software while handling several conflicting objectives (i.e, isolation, cost). We have integrated *Sputnik* in the Polymer[1] framework and evaluated it using Kevoree[2] model@run.time platform. We have conducted experiments to compare natural selection performance versus *Sputnik* performance. Our experiments highlight that *Sputnik* results in a faster convergence by reducing the number of generations while conserving the ability to achieve acceptable trade-offs in our use case. This paper is organized as follows. Section 2 describes the key concepts related to this paper. Section 3 presents *Sputnik* hyper-heuristic. Section 4 presents validation elements of our approach. Finally, Sections 5 and 6 discuss the related work, our conclusion and future work.

II. Cloud Optimization, MOEAs Run-time Constraints and Hyper-heuristics

This section introduces existing approaches that have tackled the cloud multi-objective optimization problem. We also briefly outline MOEAs concepts used in *Sputnik* and their relevance to drive run-time optimization. In a second step, we give an overview about MOEAs performance issues with a particular focus on run-time usage. Finally, we highlight the role of hyper-heuristics to improve MOEAs algorithms efficiency.

A. Multi-Objective Optimization of Software Deployment in the Cloud

When moving their applications to the cloud, cloud customers take advantage of an elastic environment in which resources are provisioned/deprovisioned automatically to adapt to variable workload. Cloud elasticity leverages a set of actions that are responsible to move cloud configurations from one state to another. This leads to a very wide set of potential candidate solutions, and consequently to a wide domain to explore in order to find at a given time the best cloud configuration. Different cloud optimization axes have been recently explored over the literature. For instance some of the approaches are cost effective, other approaches are oriented towards performance improvement or security hardening while others are oriented towards achieving an eco-friendly green cloud. In [9], the authors have defined an algorithm called optimal cloud resource provisioning (OCRP) in which they provide an optimal cloud resources provisioning by formulating and solving a stochastic integer programming. The results show that the approach is able to reduce resources costs. In [23], the authors have proposed Mistral, a controller framework that optimizes power consumption, performance, and costs. Mistral is based on a search algorithm that takes into consideration the costs induced by the search algorithm itself. In [24], we study how security, isolation requirements and performance objectives can be considered simultaneously in the same optimization process.

B. MOEAs Concepts

As depicted by [38], genetic based approaches are suitable candidate for scheduling and planning problems. Genetic Algorithms (GA) are driven by elitism rules that favor the survival of strongest species (best candidate solutions) in analogy to natural selection [36]. They are based on an iterative search

[1] http://kevoree.org/polymer/

[2] http://kevoree.org/

process, which involves a set of individuals that are randomly selected and mutated (or mixed using crossover operator) in each iteration to constitute the next *generation* population. *Fitness functions* [4] are used to evaluate solutions with regards to a specific optimization problem, in analogy to natural selection where species qualities are evaluated according to their surrounding context. Genetic algorithms introduce changes in the population to create new individuals called offspring through the following operators [35]:

- The *crossover* operator generates offspring by a genetic recombination of the two selected parents. The resulting offspring maintains some features from each parent, thus maintaining population diversity.
- The *mutation* operator introduces small changes on an individual with a probability to improve the population diversity [28].
- The *selection* operator selects a fixed number of fittest offspring for the next generation to maintain a fixed population size and puts good offspring into the next generation with a high probability.

More formally, a multi-objective optimization aims at *minimizing* a vector function $F(x) = (f_1(x), f_2(x), ..., f_n(x))$ *where* $x \in \Re_n$ and F is a vector of n objective functions. Multi-Objective optimization introduces two important concepts: The *Pareto Dominance* and the *Pareto Optimality* [12]: given two solution X_1 and X_2, X_1 is said to *dominate* X_2, ($X_1 \succeq X_2$), if $f_i(X_1) \preceq f_i(X_2)$, for all $i = 1,..,n$ and $f_i(X_1) \prec f_i(X_2)$ for at least one objective function $f_i(X)$. A Pareto-optimal front is a curve that groups all solutions belonging to the Pareto-optimal set.

Multi-Objective Evolutionary Algorithms (MOEAs) have been successfully applied in many domains such as finance, logistics, test cases optimization [2] and recently in cloud engineering problems. In [10], the authors have provided a taxonomy of the different application domains of MOEAs. Some of these applications have to be used in a run-time context where dynamic parameters adjustment is required such as running vehicles guidance. Run-time optimization has thus to be performed with constrained resources to cope with run-time constraints, the next section highlights these constraints.

C. Performance of MOEAs at run-time

The usage of MOEAs in run-time optimization problems [21], for instance load balancing problems which can be seen as a subset of scheduling problems, motivates the need to improve their performance. Several studies have explored the computational costs of MOEAs [26], [36] by evaluating their performance on different problems using various MOEAs categories. According to [22], MOEAs computational costs can be reduced by reducing its algorithmic complexity or the computational costs of the fitness function. Several studies have been focusing on highlighting the computational costs of fitness evaluations while proposing models to reduce its cost. In [6], the authors have tested complex fitness functions, and have proposed the concept of surrogate models to reduce their computational cost. Their approach is based on a gaussian optimization model that evaluates previous fitness functions to estimate future fitness functions scores instead of evaluating real fitness functions. In [34], authors highlight the impact of large populations on the computation time of the Pareto front [20], [19]. In [26], authors conduct experiments and highlight the efficiency loss and overhead introduced by a number of objectives above 3.

All of these studies highlight MOEAs performance drawbacks and propose specific solutions to improve MOEAs. In this paper, we focus on the notion of hyper-heuristic (described in the next section) to propose a solution with an impact on the algorithm complexity.

D. Hyper-heuristics: Classification and Objectives

In search based engineering, hyper-heuristics [7] define methods that act on adapting search parameters over the search process. Hyper-heuristics introduce modifications on the algorithm itself, in order to improve its computational efficiency or effectiveness to handle a specific purpose [33]. Hyper-heuristics rely on machine learning mechanisms [7], to leverage knowledge assessed during each search iteration. For example, in [1] such learning methods store neighborhood information through a neural network to improve a genetic algorithm accuracy.

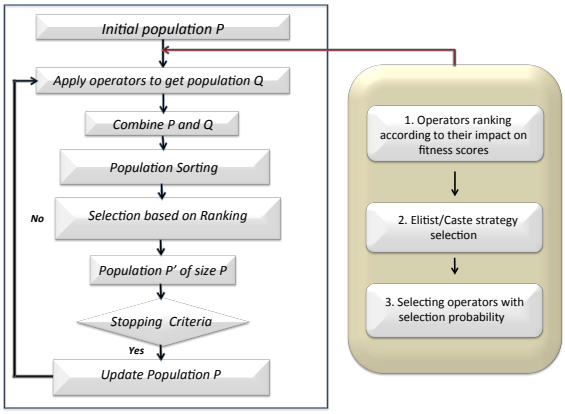
Hyper-heuristics can be classified according to the following taxonomy [7] depending on the nature of the heuristic used (based on selection or generation methodologies). For each nature, we thus differentiate : (i) *Online learning* hyper-heuristics which learn while the search algorithm is running, (ii) *Offline learning* hyper-heuristics which learn from the system before the execution of the search process. The contribution of this paper, falls into the category of hyper-heuristics that embed online learning mechanisms which are based on both selection and generation methodologies to achieve performance improvement over a set of software engineering problems [17], [29]. Indeed, our approach relies on mutation operator prioritization (selection nature) based on the evaluation of past execution efficiency (generation nature).

III. *Sputnik*: An Hyper-heuristic for an efficient self-adaptive systems optimization

The randomness introduced by natural selection of evolutionary approaches leads to suboptimal performance in terms of computational power and memory usage. Random selection of mutation operators produces useless candidate solutions that lead to computational resources wastage and therefore does not meet run-time optimization constraints. In modern biological studies, after identifying a gene impact on an individual phenotype trait, scientists like Muller *et al* [31] leverage artificial mutation to directly produce an individual combining the foreseen modification.

Our hypothesis is that the artificial mutation concept can be introduced in evolutionary algorithms to mimic modern biological genetics, thus reducing the number of required generations to reach acceptable solutions. The *a priori* scientific knowledge of a gene modification impact, could be replaced by a continuous ranking and learning approach leveraging execution history. Therefore, in this paper we aim at optimizing MOEAs,

Fig. 1: *Sputnik* Workflow



by dynamically reducing the usage of mutation operators that are less effective in improving fitness functions scores. At the same time, we maintain the equity of natural selection, to ensure that the modified evolution algorithm is able to reach any solution. Thus, we replace the random mutation operator selection by an hyper-heuristic that detects for each individual the most pertinent operator to apply in order to achieve a faster trade-off.

After each application, mutation operators are classified according to the delta variance they introduce on each fitness function. Internally, *Sputnik* maintains an elitist group of mutation operators that are relevant to improve a fitness function score. To enhance operators selection, *Sputnik*, considers the current fitness scores reached by a solution, and selects the most relevant mutator in elite groups to improve next generation[3]. As illustrated in Figure 1, *Sputnik* takes as inputs an initial population and a generation number. As most of MOEAs variants like (NSGA-II, ε-MOEA, SPEA 2 [36]), the algorithm is based on an individual ranking step according to fitness function evaluation and a non dominating population construction. *Sputnik* introduces a favoritism operator approach in the mutation process described as follows:

We consider a multi-objective evolutionary optimization of f with n objectives $(f_1, f_2,...,f_n)$. The average fitness score for a generation g_l is defined by $\sum_{i=1}^{n} f_i/n$ for each individual in the generation. We define $\triangle_{impact} f_{g_l,op}$ as the fitness score variation between the average of fitness function evaluation for a generation g_{l-1} and a generation g_l that is achieved by the operator *op*:

$$\triangle_{impact} f_{g_l,op} = (\sum_{i=1}^{n} f_i/n)_{g_l,op} - (\sum_{i=1}^{n} f_i/n)_{g_{l-1},op}$$

Sputnik records the *selection occurrence* for the different mutation operators that have been involved over the search process. Once all mutation operators have been selected at least once, $\triangle_{impact} f_{g_l,op}$ is evaluated for all the operators and *Sputnik* is configured to select the operators that have $\triangle_{impact} f_{g_l,op} > 0$ in the generation g_{l+1} with the Elitist or the Caste strategies. More formally, *Sputnik* selection function $selection_{op}$ is specified in Figure 2:

$P_{selection}$ depends on *Sputnik* strategy and is evaluated as follows:

- *Elitist Strategy*: The operator that has the highest $\triangle_{impact} f_{g_l,op}$ is selected in the generation g_{l+1} with a high $P_{selection}$. This configuration accords higher chance to the "winner operator" to be selected in the next generation. All others operators are selected with a probability 1-$P_{selection}$.
- *Caste Strategy*: A selection probability is partitioned between operators which have $\triangle_{impact} f_{g_l,op} \preceq 0$ and is defined as follows:
  $P_{selection} = \triangle_{impact} f_{g_l,op} / \sum_{op \in operators} \triangle_{impact} f_{g_l,op}$.
  This configuration gives more equity in terms of selection probability for all operators which have a positive impact on a fitness score.

Sputnik-based mutation operators selection is described in Algorithm 1. In both settings, we set a selection probability of 10% for pure random selection of operators to not discriminate worst ranked operators. The random selection of operators mimics the natural evolution and aims at giving equitable chances to all solutions, and to any potential mutation operator. This random operators selection ensures a proper exploration of the domain and prevents the solutions to fall into a local minimum. *Sputnik* keeps 10% of mutation to give chance to less selected operators to be reintroduced in the elite group of a fitness function. Through this mechanism, we keep a minimal equity of species while conserving 90% of the Pareto for most efficient mutation.

IV. Validation

To evaluate *Sputnik*, we have considered an experimental scenario in which a cloud provider aims at placing several software components related to many customers in the different Virtual Machines running on the top of physical machines. We define a cloud configuration as an architecture model which leverages virtual machines and components (i.e, provisioned software in our context) concepts. Based on our architectural model, an *Individual* represents a solution vector *X* that corresponds to a cloud infrastructure model. The reader may refer to our open source Polymer framework which gives ample details about our architectural model[4]. A *gene* corresponds to a component, a virtual machine or a physical machine in our model. A *population* corresponds to a set of cloud infrastructure models. A *genetic mutation operator* corresponds to an elementary flip in the model that is introduced by an elementary operation. A Cloud infrastructure multi-objective optimization problem is represented by the following Triplet *(I,F,CO)*. I denotes a cloud infrastructure model which represents an abstraction of a set of (VM). Each (VM) hosts *n* software components (C). *CO* denotes a set of possible configurations in *I* that satisfy F. A configuration $co \in CO$ is obtained through a mapping from Components (C) to Virtual Machines (VMs),

[3]http://www.genetics.org/content/111/1/147.short

[4]http://kevoree.org/polymer/

Fig. 2: *Sputnik* selection function

$$\begin{array}{rcl} selection_{op} : operators \times generation \times objectives & \longrightarrow & operator \times probability \\ (op_1, op_2, ..., op_m),\ g_l,\ (f_1, f_2, ..., f_n) & \longmapsto & op_{max(\triangle_{impact} f_{g_l,op})} \times P_{selection} \end{array}$$

Algorithm 1 Sputnik

Input: Population P, Generation Number g, Operators Op_{set}, List of operators operator-used=$\varnothing$, boolean *sputnik_active=false*
Output: Population P

```
Apply randomly a mutation operator Op_current on P to get P_new
for all j where j ranges from 1 to g do
  Evaluate f_i(x) on P
  operator-used:=operator-used ∪ Op_current
  /* Sputnik is active only if all the mutation operators have
  been at least chosen once */
  if operator-used ⊆ Op_set then
    sputnik_active=true
    Evaluate △_impact f_{g_i,op} ∀ op in Op_set
    Select P_selection ∈ {P_elitist, P_cast}
    Identify Op_best
    Select Op_best with P_selection, Op with 1-P_selection from
    Op_set and get P_new
  else
    Select Op randomly from Op_set and get P_new
  end if
  Update P based on dominance ranking and crowding
  distance to get P_new
end for
```

for example $co = (VM_1(c_1, c_2, c_3))$ denotes a configuration with a single virtual machine VM_1 hosting 3 components c_1, c_2, c_3. The vector F(X) is composed of the following 4 objective functions, $F(X) = (f_1(x), f_2(x), f_3(x), f_4(x))$ that have to be minimized. These objectives reflect the different axis of optimization that the provider aims at achieving when placing the software components at the level of the different Virtual Machines and are defined as follows:

- $f_1(x)$= Cost(x): Denotes Virtual Machines cost which is proportional to the number of active VMs on the top of a PaaS.
- $f_2(x)$= Isolation(x): This function is incremented by 1 whenever two components from different cloud customers share the same Virtual Machine. To achieve isolation, a cloud provider aims at hosting software components belonging to different customers workloads in different Virtual Machines.
- $f_3(x)$= Similarity(x): The similarity function quantifies the similarity between the components hosted in Virtual Machines to assess software diversity. Software diversity [3] is an indicator of potential cascading failure to quantify cloud fault tolerance capabilities.
- $f_4(x)$= Redundancy(x): The redundancy function provides a score based on redundant software (*i.e.* number of replicates of the same service).

All fitness values have been normalized to range in the interval [0,1]. Table I presents our set of operators O including 7 mutation operators and 1 crossover (SwitchOperator) operator. Software deployment in the cloud is a multi-objective optimization problem that aims at finding a cloud configuration $co \in CO$ such as min $F(X)$. We have implemented an optimization prototype in the Polymer framework which leverages a model based encoding to perform MOEAs optimization. Model@run.time paradigm [5] enables our cloud models to be seamlessly deployed in a real large-scale production environment [14], [15] like a cloud infrastructure. Thus this validation section aims at evaluating the performance improvement achieved by *Sputnik* hyper-heuristic, in terms of efficiency and effectiveness to solve the software deployment optimization problem.

A. Research Questions

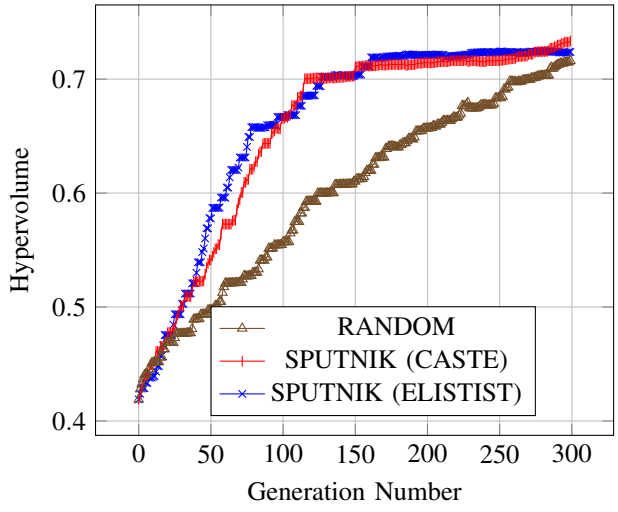
This validation section aims at exploring *Sputnik* efficiency to improve MOEAs convergence speed to achieve a certain level of trade-off between several objectives. Thus, we compare the efficiency achieved with and without *Sputnik* hyper-heuristic. We also explore the effectiveness of *Sputnik* once embedded in most popular MOEAs algorithms such as ε-MOEA and NSGA-II. As a metric to compare the solutions of the different algorithms under study, we choose the hypervolume [39] metric as Pareto-front quality indicator that defines the total size of space dominated by the solutions in the Pareto-front. Our validation steps are summarized in the following research questions:

- RQ_1 : ***Sputnik* Efficiency:** 1) Considering that an acceptable trade-off is 90% of the best solutions, is the Darwin *Sputnik* operator selection strategy successful to reduce the number of generations to reach the defined acceptable trade-off compared to a classical random strategy? 2) What is the gain in terms of execution time of *Sputnik* compared to MOEAs that are configured without *Sputnik*? 3) How does *Sputnik* perform with different probability selection values? 4) What is *Sputnik* impact on the different objectives functions that have been chosen?
- RQ_2 : ***Sputnik* Effectiveness:** Does *Sputnik* produce comparable results in terms of trade-offs achieved compared to classical random mutation selection even with modifying the equity of operators selection?
- RQ_3 : **Generalization:** What are the applicability limits of *Sputnik*? How does *Sputnik* behave with different ob-

TABLE I: Operators Definition

Operators	Description
AddVMMutator(VM_i,PaaS)	Creates a Virtual node VM_i on the top of a PaaS
AddSoftwareMutator(S,VM_i,PaaS)	Creates a component S in the Virtual node VM_i
CloneNodeMutator(VM_i,PaaS)	Creates a clone of VM_i on the top of PaaS
RemoveNodeMutator(VM_i,PaaS)	Removes a Virtual node VM_i on the top of a PaaS
RemoveSoftwareMutator(S,PaaS)	Removes a software component S from the PaaS
AddSmartMutator(S,PaaS)	Adds a software component S from the Virtual Node that contains the least number of components
RemoveSmartMutator(S,PaaS)	Removes a component S from the Virtual Node that contains the largest number of components
SwitchOperator(S_1,VM_1,S_2,VM_2,PaaS)	Switches the component S_1 from the Virtual node VM_1 to VM_2 and switches S_2 from the VM_2 to VM_1

Fig. 3: Hypervolume: *Sputnik* (Elitist & Caste) versus Random Selection



jectives? How does it behave with the different variants of MOEAs?

B. Experimental results

To answer RQ_1, we embed *Sputnik* in a cloud optimization engine that manages 100 virtual nodes and maintains a web front-end and a load-balancer software components that have to be dispatched in the different Virtual Machines (VMs). Our cloud reasoning engine is configured to leverage an ϵ-NSGA II [12]. We perform 30 runs of our experiments with 5 populations and 300 generations using the following configurations: *Sputnik* with Caste Strategy, *Sputnik* with Elitist Strategy and finally with random operators selection. The average hypervolume values of the results obtained in the 30 runs, are depicted in Figure 3 according to the generation number and in Figure 4.a according to the elapsed time. We consider that a solution achieves an acceptable trade-off if it reaches 90% of the best obtained solution. In our case study, the best obtained solution achieves an hypervolume of 0.79 (acceptable hypervolume value is 0.71 in our case), it has been reached with a 250 generations previous run. A run with *Sputnik* (with both strategies) reaches this value after 176 generations whereas a run with random operators selector reaches this value after 279 generations, respectively 8s for *Sputnik* and 16s for the random selection. *Sputnik* strategies are very similar in terms of hypervolume achievements, they both reach a value around 0.76. We notice that elitist strategy converges slightly faster however, the caste strategy can reach better hyper-volume scores. These results can be justified by mutators diversity introduced by the caste strategy, which favors at a certain extent operators equity. In average, *Sputnik* (both with caste and elitist configurations) outperforms random selection by reducing around 37% the number of necessary generations, and around 50% the time to reach acceptable solutions. This confirms our first hypothesis which states that a smart mutation selection strategy is successful to improve efficiency to reach acceptable trade-offs compared to a classical random selection strategy.

To explore the impact of the selection probability of the elitist strategy on *Sputnik* efficiency, we run the same previous experiment with two different probability values of $P_{selection}$. The results of a selection probability of 90% and 50% are shown in Figure 4.a and Figure 4.b. Unsurprisingly, we observe that the more *Sputnik* uses its learning strategy, the best is the convergence speed comparing to a random selection.

We also have evaluated the values reached by the different objectives of our case study (Cost, Redundancy, Similarity, Isolation). The results are illustrated in Figure 5. The chart presents the cost per hour and SLA satisfaction percentage reached for the Redundancy, Similarity, Isolation objectives. Note that the values obtained for the mono-objective optimization correspond to distinct runs in which we aimed at optimizing one objective at once on the detriment of other objectives. For our minimization multi-objective optimization problem, *Sputnik* achieves 3 better objectives values in 400 generations compared to a standard NSGA-II: For the different objectives (Cost, Similarity, Redundancy, Isolation), a

Fig. 4: NSGA-II Hypervolume with different probabilities of *Sputnik* Selector

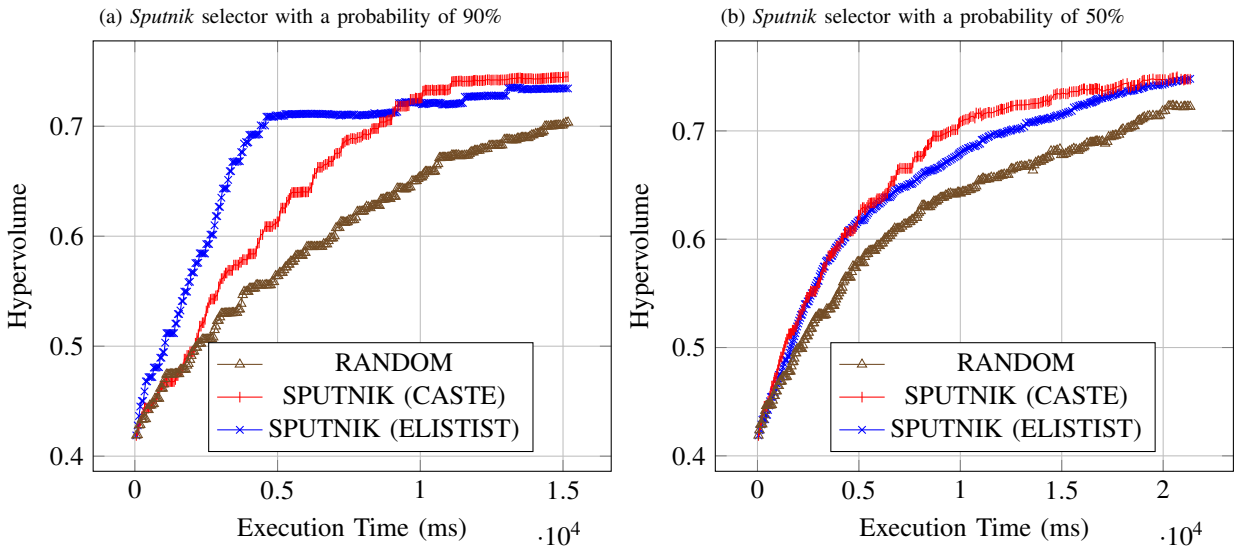
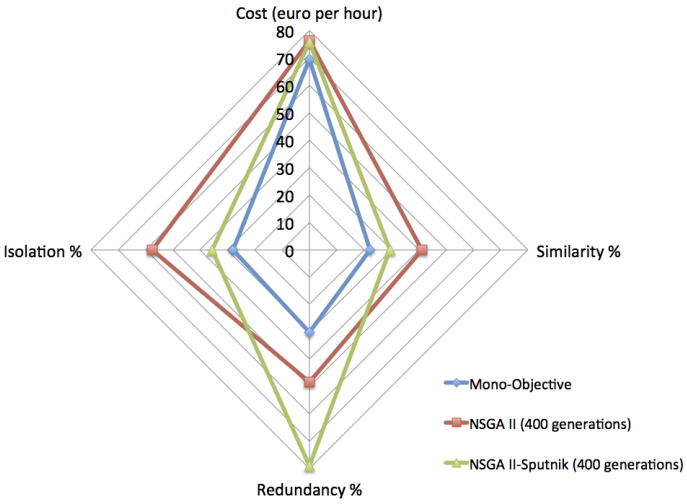


Fig. 5: Objectives Chart Radar



possible solution presents the following values (75.96, 29.41, 78.94, 35.57) compared to (76.41, 41.26, 48.38, 57.54) for standard NSGA-II. We conclude that for the same number of generations, we obtain results that exhibit better trade-offs with *Sputnik* activated on top of NSGA-II.

To answer RQ_2, we run a similar experiment with both random and *Sputnik* selector until we reach an unchanged value of hypervolume over 50 generations. Final values for *Sputnik* (elitist: 0.77, caste: 0.81 and random 0.78 allow us to conclude that our hyper-heuristic does not decrease the quality of the results in terms of degree of trade-off achieved. Moreover the caste strategy improves the hypervolume score.

To answer RQ_3, we have evaluated *Sputnik* with different MOEAs algorithms and various number of objectives. Given that several factors (i.e., implementation aspects, existence of other processes running in the machine, etc) may influence execution time of our approach, we have compared the hypervolume reached over 400 generations. The results are shown in Figure 6 with hypervolume distribution in Figure 7. For NSGA-II algorithm, an hypervolume value of 0.7 is reached after 90 generations with Elitist strategy, 120 generations with Caste strategy, and reached after 240 generations with random strategy. A *Sputnik* on top of ε-NSGA-II for almost both the Caste and the Elistist version achieves an hypervolume of 0.6 in 100 generations. Similarly to NSGA-II based approaches, we observe a similar speedup for SMS-MOEA and ε-MOEA algorithms. Above 300 generations for NSGA-II and ε-NSGA-II, we also observe that the *elitist* strategy could lead to little decreased effectiveness. This result could be explained by the total order between operators introduced by *elitist* strategy, which reduces diversity. We conclude that the *caste* strategy is less intrusive hyper-heuristic which maintains better the algorithm effectiveness. From these runs, we notice that the *Sputnik* hyper-heuristic can be generalized on several MOEAs.

Secondly, we have explored the generalization of *Sputnik* with variable objectives by analyzing the hypervolume while varying the objectives from 1 to 4. The results, generated with NSGA-II for 200 generations are presented in Figure 8 in term of optimization time. We observe that *Sputnik* with the caste and elitist strategies provides faster hypervolume convergence compared to random with smaller number of objectives. Indeed for 1 and 3 objectives, the *Sputnik* strategy selects efficient operators faster, however above 4 objectives, *Sputnik* effect tends to be similar to random selection. We explain such effect by the fact that *Sputnik* does not keep the history of previous selected operator selection, thus above 4 objectives, the selection tends to be random. In future work we plan to

Fig. 6: Hypervolume over 400 generations

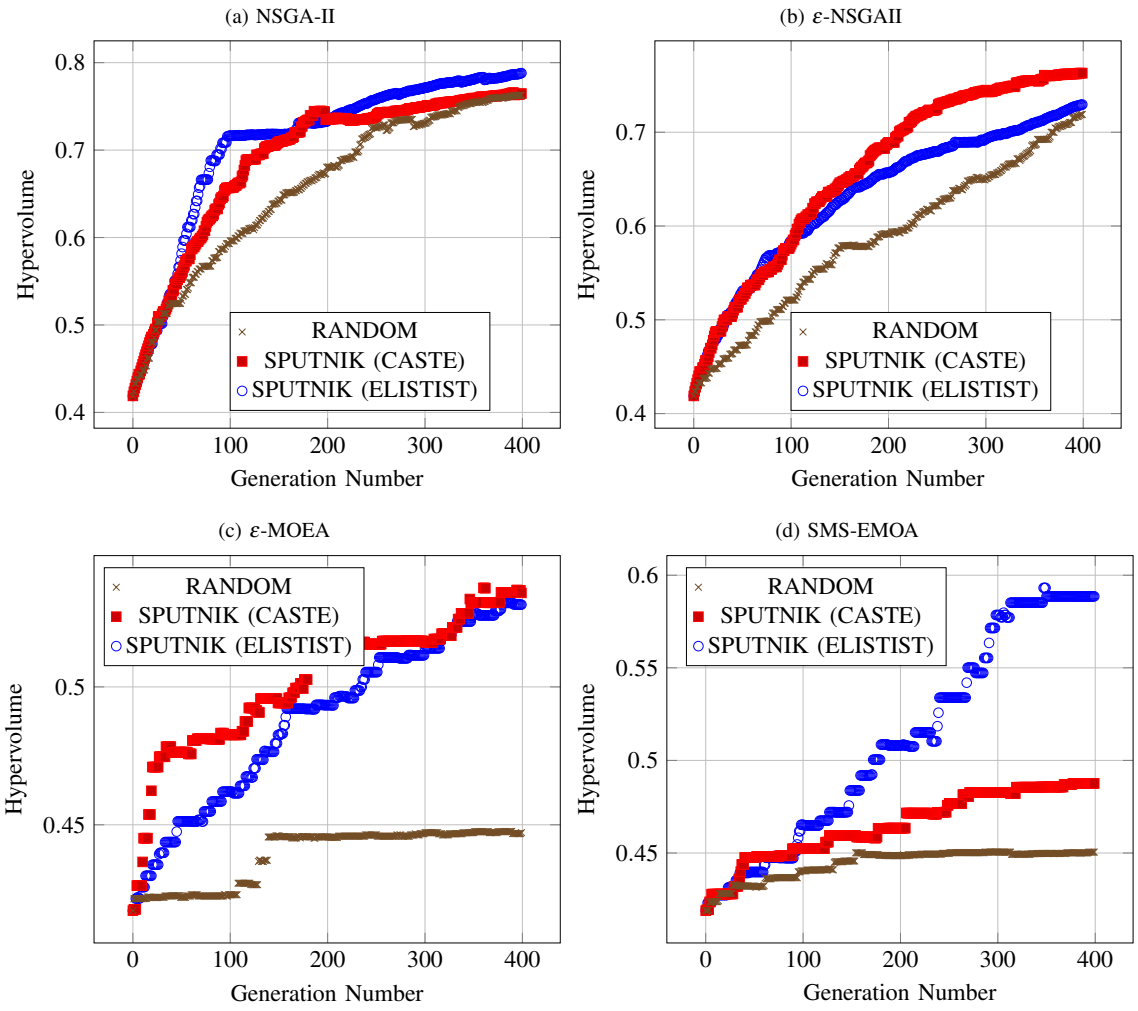


Fig. 7: NSGA-II, ε-NSGAII, ε-MOEA, SMS-EMOA Box Plots

Statistical Distribution for the hypervolume over 400 generations shown above: Max values reached with *Sputnik* with its two settings outperform max values reached with plain NSGA-II and ε-NSGA-II. ε-MOEA, SMS-EMOA have weaker results due to their selection strategy (one mutation per generation) that slows down *Sputnik*.

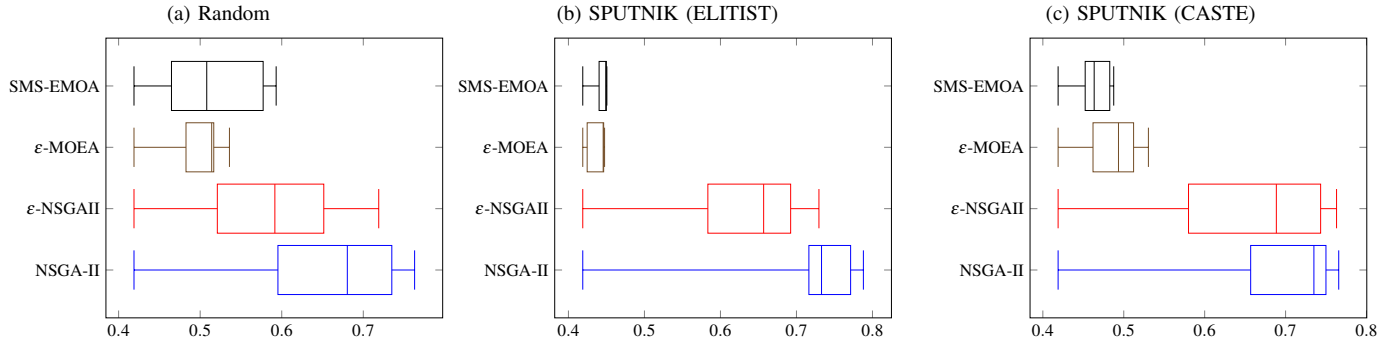
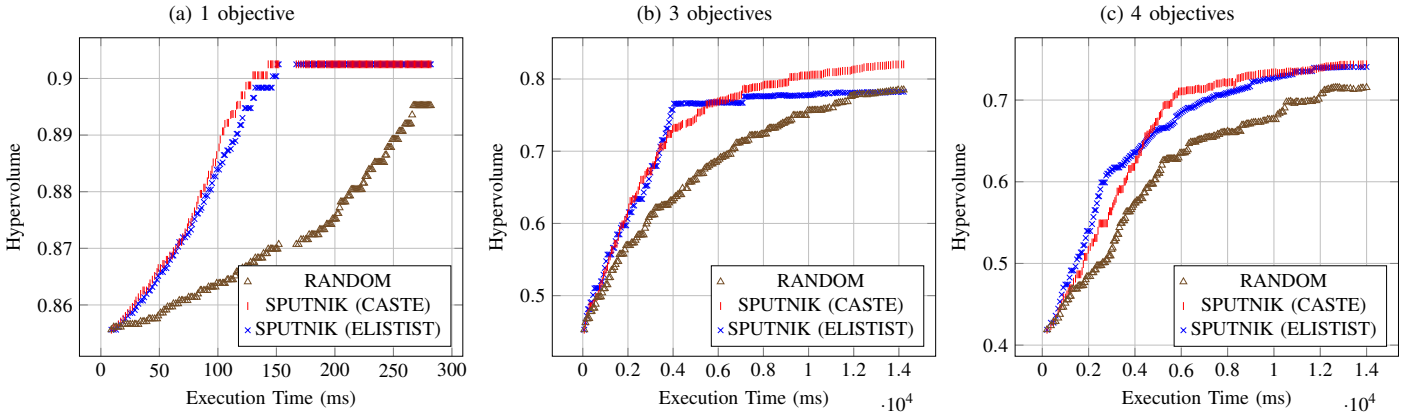


Fig. 8: NSGA-II hypervolume over Execution Time with Variable Objectives



add operators selection history to maintain *Sputnik* efficiency independently of the objectives number.

C. Threats to Validity

Internal validity is related to the parameters setting of our experiments (i.e. generation number, objectives number, population size, operators, etc.), such as values chosen for the value of ε in the ε-dominance which might impact our validation results. More specifically, *Sputnik* incrementally builds a mapping of each operator impact on a particular fitness function. Thus, the coupling effect between the eight used operators and a particular fitness function could introduce a bias on our results. To minimize this bias, we leverage a set of operators without direct coupling effect.

Construct validity arises from the bias introduced in the way we build our experiments. In each experiment presented in the paper, we have compared one run of *Sputnik* against a random mutator selector. As for any random based techniques, a set of repeated experiments should be run to draw statistically significant results. This bias is mitigated by the number of different experiments that have been run on *Sputnik* which all demonstrate the effectiveness of the approach.

External validity is related to the generalization of observed results outside the case study presented in the validation section. Although we applied the approach to a cloud reasoning engine and have shown the impact of *Sputnik* to improve the efficiency of the optimization, further experimentation is needed on different systems that have to comply with run-time constraints in their optimization process.

V. Related Work

This work focuses on hyper-heuristics that operate on top of MOEAs to improve their usage in a run-time context and particularly to a cloud-based deployment software. In [33], the authors embed learning techniques in classical MOEAs. They assume that objective functions are expensive to compute so they rank the Pareto front elements and they evaluate only the individuals that have higher ranks. In [27], the authors have proposed an hyper-heuristic that relies on the hypervolume calculation to improve computational results. These approaches consider only the Pareto front set evaluation to improve MOEAs, whereas our approach evaluates operators contribution in improving fitness and thus injects mutation operators that are eligible to make MOEAs converge faster. In [37], the authors have shown that racing algorithms can be used to reduce the computational resources inherent from using evolutionary algorithms in large scale experimental studies, their approach automates solutions selection and discards solutions that do not introduce results improvement. Whereas racing techniques eliminate worst solutions candidates to speed up the search, in our approach we keep considering worst ranked candidates to maintain operators diversity. In [13], the authors have explored the advantages of using a controlled crossover on top of single-point search based hyper-heuristics. They maintain the best solutions obtained during the search and update crossover operator accordingly. The authors rely on a process focused on crossover as a biological selective breeding. This breeding assumes that fittest genes are already present in the initial population. Unlike cited approaches, *Sputnik* focuses on artificial mutation selection, therefore mimicking the process used to produce genetically modified organisms. As far as we know, there is no other hyper-heuristic that proposes artificial mutation at mutation operators level.

VI. Conclusion

In this paper, we have introduced a hyper-heuristic breaking the random natural selection of classical MOEAs to leverage an elitist artificial mutation inspired by biological studies [31]. *Sputnik* relies on a mutation operator selection based on a continuous learning of past effect on fitness functions, instead of random mutation operators selection. The overall goal of *Sputnik* is to enhance the optimization algorithm itself, and to guide the search towards faster trade-offs achievement to finally save generation cycles and time. Experimentally, we provide evidence of the effectiveness of artificial mutation to reduce significantly the number of necessary generations to

find acceptable trade-offs. In the future, we plan to explore the effectiveness of *Sputnik* in accelerating the optimization processes of other adaptive systems that have to comply with run-time constraints.

VII. Acknowledgement

This research is supported by the Fonds National de la Recherche, Luxembourg C12/IS/4011170 in the context of TOOM Core project.

References

[1] Deriving operating policies for multi-objective reservoir systems: Application of self-learning genetic algorithm. *Applied Soft Computing*, 10(4):1151 – 1163, 2010.

[2] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Automatic test case optimization using a bacteriological adaptation model: application to. net components. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 253–256. IEEE, 2002.

[3] B. Baudry, M. Monperrus, C. Mony, F. Chauvel, F. Fleurey, and S. Clarke. DIVERSIFY - Ecology-inspired software evolution for diversity emergence. In *CSMR*, pages 395–398. IEEE, 2014.

[4] D. Beasley, R. Martin, and D. Bull. An overview of genetic algorithms: Part 1. fundamentals. *University computing*, 15:58–58, 1993.

[5] G. Blair, N. Bencomo, and R. B. France. Models@ run. time. *Computer*, 42(10):22–27, 2009.

[6] D. Buche, N. Schraudolph, and P. Koumoutsakos. Accelerating evolutionary algorithms with gaussian process fitness function models. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, pages 183–194, 2005.

[7] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward. A classification of hyper-heuristic approaches. In *Handbook of Metaheuristics*, pages 449–468. Springer, 2010.

[8] I. Cases and V. de Lorenzo. Genetically modified organisms for the environment: stories of success and failure and what we have learned from them. *International microbiology*, 8(3):213–222, 2010.

[9] S. Chaisiri, B.-S. Lee, and D. Niyato. Optimization of resource provisioning cost in cloud computing. *Services Computing, IEEE Transactions on*, pages 164–177, 2012.

[10] C. A. C. Coello and G. B. Lamont. *Applications of multi-objective evolutionary algorithms*, volume 1. World Scientific, 2004.

[11] C. Darwin. *On the origins of species by means of natural selection*. 1859.

[12] K. Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[13] J. H. Drake, E. Özcan, and E. K. Burke. Controlling crossover in a selection hyper-heuristic framework. *School of Computer Science, University of Nottingham, Tech. Rep. No. NOTTCS-TR-SUB-1104181638-4244*, 2011.

[14] F. Fouquet, E. Daubert, N. Plouzeau, O. Barais, J. Bourcier, and J.-M. Jézéquel. Dissemination of reconfiguration policies on mesh networks. In *Distributed Applications and Interoperable Systems*, pages 16–30. Springer, 2012.

[15] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel. A dynamic component model for cyber physical systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pages 135–144. ACM, 2012.

[16] S. Frey, F. Fittkau, and W. Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 512–521, 2013.

[17] I. Güney, G. Küçük, and E. Özcan. Hyper-heuristics for performance optimization of simultaneous multithreaded processors. In *Information Sciences and Systems 2013*, pages 97–106, 2013.

[18] M. Harman, K. Lakhotia, J. Singer, D. R. White, and S. Yoo. Cloud engineering is search based software engineering too. *Journal of Systems and Software*, 86(9):2225–2241, 2013.

[19] H. Ishibuchi, Y. Nojima, and T. Doi. Comparison between single-objective and multi-objective genetic algorithms: Performance comparison and performance measures. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 2006.

[20] H. Ishibuchi, Y. Sakane, N. Tsukamoto, and Y. Nojima. Evolutionary many-objective optimization by nsga-ii and moea/d with large populations. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pages 1758–1763, Oct 2009.

[21] M. T. Jensen. Reducing the run-time complexity of multiobjective eas: The nsga-ii and other algorithms. *Evolutionary Computation, IEEE Transactions on*, 7(5):503–515, 2003.

[22] Y. Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft computing*, 9(1):3–12, 2005.

[23] 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 *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 62–73, 2010.

[24] D. E. Kateb, F. Fouquet, G. Nain, J. A. Meira, M. Ackerman, and Y. L. Traon. Generic cloud platform multi-objective optimization leveraging models@run.time. pages 343–350, 2014.

[25] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 2003.

[26] V. Khare, X. Yao, and K. Deb. Performance scaling of multi-objective evolutionary algorithms. In *Proceedings of the 2Nd International Conference on Evolutionary Multi-criterion Optimization*, EMO'03, pages 376–390, 2003.

[27] C. León, G. Miranda, and C. Segura. Hyperheuristics for a dynamic-mapped multi-objective island-based model. In *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, pages 41–49. 2009.

[28] S. W. Mahfoud. Niching methods for genetic algorithms. 1995. UMI Order No. GAX95-43663.

[29] E. Özcan, B. Bilgin, and E. E. Korkmaz. A comprehensive analysis of hyper-heuristics. *Intell. Data Anal.*, pages 3–23, 2008.

[30] P. Patel, A. H. Ranabahu, and A. P. Sheth. Service level agreement in cloud computing. 2009.

[31] H. Plaskett. Artificial transmutationof the gene. *tic*, 1927.

[32] C. R. Reeves and J. E. Rowe. *Genetic algorithms: principles and perspectives: a guide to GA theory*, volume 20. Springer, 2003.

[33] C.-W. Seah, Y.-S. Ong, I. W. Tsang, and S. Jiang. Pareto rank learning in multi-objective evolutionary algorithms. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8, 2012.

[34] K. C. Tan, T. H. Lee, and E. F. Khor. Evolutionary algorithms with dynamic population size and local exploration for multiobjective optimization. *Evolutionary Computation, IEEE Transactions on*, 5(6):565–588, 2001.

[35] D. A. Van Veldhuizen. *Multiobjective evolutionary algorithms: classifications, analyses, and new innovations*. PhD thesis, 1999.

[36] D. A. Van Veldhuizen and G. B. Lamont. Multiobjective evolutionary algorithms: Analyzing the state-of-the-art. *Evolutionary computation*, 8(2):125–147, 2000.

[37] B. Yuan and M. Gallagher. Statistical racing techniques for improved empirical evaluation of evolutionary algorithms. In *Parallel Problem Solving from Nature-PPSN VIII*, pages 172–181, 2004.

[38] A. Zhou, B.-Y. Qu, H. Li, S.-Z. Zhao, P. N. Suganthan, and Q. Zhang. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, pages 32–49, 2011.

[39] E. Zitzler, D. Brockhoff, and L. Thiele. The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration. In *Evolutionary Multi-Criterion Optimization*, pages 862–876, 2007.