Queuing Theoretic and Evolutionary Deployment Optimization with Probabilistic SLAs for Service Oriented Clouds

Hiroshi Wada and Junichi Suzuki
Department of Computer Science
University of Massachusetts, Boston
Boston, MA 02125-3393
{shu, jxs}@cs.umb.edu

Katsuya Oba
OGIS International, Inc.
San Mateo, CA 94404
oba@ogis-international.com

Abstract

This paper focuses on service deployment optimization in cloud computing environments. In a cloud, each service in an application is deployed as one or more service instances. Different service instances operate at different quality of service (QoS) levels. In order to satisfy given service level agreements (SLAs) as end-to-end QoS requirements of an application, the application is required to optimize its deployment configuration of service instances. E^3/Q is a multiobjective genetic algorithm to solve this problem. By leveraging queuing theory, E^3/Q estimates the performance of an application and allows for defining SLAs in a probabilistic manner. Simulation results demonstrate that E^3/Q efficiently obtains deployment configurations that satisfy given SLAs.

1 Introduction

This paper envisions service-oriented applications in cloud computing environments. A service-oriented application consists of a set of *service*s and a *workflow*. Each service encapsulates the function of an application component. Each workflow defines how services interact with each other. When an application is uploaded to a cloud, the cloud deploys each service in the application as one or more service instances. Each service instance runs on a process or a thread and operates based on a particular deployment plan; different service instances operate at different quality of service (QoS) levels. For example, Amazon Elastic Compute Cloud (EC2)[1] offers five different deployment plans that allow service instances to yield different QoS levels by providing different amounts of resources at different prices[2]. If an application is intended to serve different categories of users (e.g., users with for-fee and free memberships), it is instantiated with multiple workflow instances, each of which is responsible for offering a specific QoS level to a particular user category.

A service level agreement (SLA) is defined upon a workflow as its end-to-end QoS requirements such as throughput, latency and cost (e.g., resource utilization fees). In order to satisfy given SLAs, application developers (or cloud engineers) are required to optimize a deployment configuration of service instances for each user category by considering which deployment plans and how many service instances to use for each service. For example, a deployment configuration may be intended to improve the latency of a heavily-accessed service by deploying its instance with an expensive deployment plan that allocates a large amount of resources. Another deployment configuration may deploy two service instances with two inexpensive deployment plans connected in parallel for improving the service's throughput.

This decision-making problem, called the SLA-aware service deployment optimization (SSDO) problem, is a combinatorial optimization problem that searches the optimal combinations of service instances and deployment plans. There exist three research issues in this problem. First, it is known NP-hard [1], which can take a significant amount of time, labor and costs to find the optimal deployment configurations from a huge search space (i.e., a huge number of possible combinations of service instances and deployment plans). The second issue is that the SSDO problem often faces trade-offs among conflicting QoS objectives in SLAs. For example, in order to reduce its latency, a service instance may be deployed with an expensive deployment plan; however, this is against another objective to reduce cost. Moreover, if the service's latency is excessively reduced for a user category, the other user categories may not be able to satisfy their latency requirements. The third issue is that traditional SLAs often consider QoS require-

[1] www.amazon.com/ec2

[2] A deployment plan with a 1.0GHz CPU and 1.7GB memory costs $0.1 per hour. Another deployment plan with four 2.0GHz CPU cores and 15.0GB memory costs $0.8 per hour.

ments as their average (e.g., average latency). This fails to consider fluctuation/variance in runtime QoS measures.

This paper proposes and evaluates an optimization algorithm, called E^3/Q, (Evolutionary multiobjective sErvice deployment configuration optimizEr), which addresses these research issues. E^3/Q is a multiobjective genetic algorithm (GA) that balances the trade-offs among conflicting QoS objectives in SLAs and seeks a set of Pareto-optimal deployment configurations that satisfy the SLAs. Given multiple Pareto-optimal configurations, application developers can better understand the trade-offs among QoS objectives and make a well-informed decision to choose the best deployment configuration for them according to their requirements and preferences. E^3/Q is practical enough to be well-applicable to any clouds that differentiate resource provision, in turn QoS, for services; for example, Amazon EC2, FlexiScale[3] and GoGrid [4]. Simulation results demonstrate that E^3/Q efficiently obtains quality deployment configurations that satisfy given SLAs by heuristically examining very limited regions in the entire search space.

2 Service Deployment and QoS Models in E^3/Q

In E^3/Q a workflow consists of a set of services. An example workflow in Figure 1 consists of four services. It has a branch after Service 1 (E^3/Q assumes that branching probabilities are known from history data), and executes Service 2 and Service 3 in parallel. In order to process requests, each service is instantiated as a service instance(s) and deployed on a particular deployment plan(s). A set of service instances and deployment plans is collectively called a deployment configuration. In Figure 2 Service 1 is instantiated as three service instances and deployed on two Deployment Plan 1 and one Deployment Plan 3. E^3/Q assumes that a deployment plan cannot operate more than two service instances of the same service but it can have instances of different services at a time. (Deployment Plan 2 in Figure 2 has two service instances.) Each deployment configuration can have arbitrary number of deployment plans and service instances to improve the service's throughput and latency.

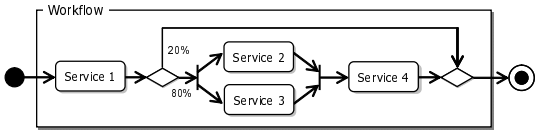

Figure 1: An Example Workflow

Figure 3 illustrates how an application processes requests from users. When a user sends a request to an application, it calls a series of service instances in its deployment configuration according to its workflow and sends back a response to the user. In this example, once receiving a request from a user an application calls one of Service 1's instances, waits for a response from the service instance, calls Service 2 and Service 3's instances in parallel, waits for responses from them, then calls a Service 4's instance. A deployment configuration is assumed to have one access point equipped with a load balancer (Figure 2). Load balancer's algorithm must be the same as that of a load balancer that an application under development uses. Currently E^3/Q supports a round robin and CPU load balancing algorithm, which dispatches requests to balance deployment plans' CPU usage. When a deployment configuration uses a round robin load balancer, requests for a certain service are dispatched to corresponding service instances with equal probability. For example, in Figure 2, when a deployment configuration receives 1,500 requests for Service 1 every second, each instance of Service 1 receives 500 requests every second since three instances are deployed in a deployment configuration.

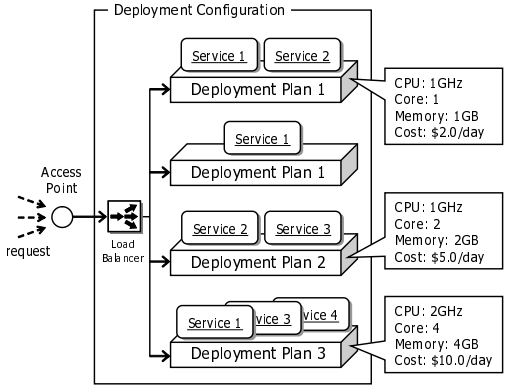

Figure 2: An Example Deployment Configuration

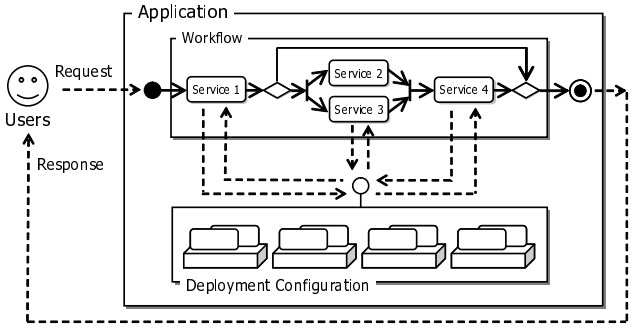

Figure 3: An Example Application

Currently, a SLA is defined with the end-to-end throughput, latency and cost of an application. In order to judge whether an application satisfies a given SLA, it is required to examine its end-to-end QoS by aggregating QoS measures of individual services. The cost of an application is

[3] www.flexiscale.com
[4] www.gogrid.com

defined as a simple summation of deployment plan's cost. Therefore, the more number and the more expensive deployment plans an application uses, the higher its cost becomes. The end-to-end throughput and latency are obtained by applying queuing theory to a deployment configuration.

2.1 Throughput and Distribution of Latency of Service Instances

Queuing theory is a well-established method for estimating performance of distributed systems and applied to a large number of research work [2, 3]. E^3/Q estimates each service instance's throughput and latency by leveraging queuing theory, and it obtains the end-to-end throughput and latency ay applying QoS aggregate functions.

Let μ_u be mean unit service rate, which is the average number of requests processed per unit time by a service instance running on a unit CPU (e.g., 1GHz CPU). μ_u is the inverse of the mean request processing time. E^3/Q assumes that μ_u of all services in a workflow are known. Based on μ_u, queuing theory can estimate throughput and the probability distribution of latency when a service instance runs on a deployment plan with various CPU configurations (e.g., one 2.5GHz CPU core or four 1GHz CPU cores) under various request arrival rate.

Assume only one service instance runs on a deployment plan with n CPU cores each of then is p times faster than a unit CPU. E^3/Q models this service instance as a $M/D/N$ queue. (Possion arrival, deterministic service and multiple service centers.) Equation 1 is an approximation of the probability that the latency (waiting time W in a queue) is greater than α [4].

$$
\begin{aligned}
Pr(W \geq \alpha) &= \left(\frac{np_n}{n-\rho}\right)\exp(-2(n-\rho)\alpha) \qquad (1)\\
\text{where } \rho &= \lambda/(p\mu_u)\\
p_n &= p_0(\rho^n/n!)\\
p_0 &= \left[\sum_{k=0}^{n-1}\frac{\rho^k}{k!} + \frac{\rho^n}{(n-1)!(n-\rho)}\right]^{-1}
\end{aligned}
$$

λ is the mean request arrival rate, which is the average number of requests to a service instance per unit time. Since μ_u is know, $Pr(W \geq \alpha)$ can be calculated when λ is given. When $\lambda > np\mu_u$, however, the usage of all n CPU cores exceeds 100%. Therefore, λ must be reduced to λ' so that $\lambda' = np\mu_u$ is hold (e.g., by dropping $\lambda - \lambda'$ requests per unit time). Hence, λ is the throughput of a service instance when $\lambda \leq np\mu_u$, while λ' (i.e., $np\mu_u$) is the throughput of a service instance when $\lambda > np\mu_u$. (λ' is also used instead of λ in Equation 1 to obtain the distribution of latency.)

When multiple service instances run on one deployment plan, portion of CPU power is assigned to each service instance based on their CPU usage. Assume instances of two services, a and b, run on a deployment plan. μ_{ua} and μ_{ub} are their mean unit service rates, and λ_a and λ_b are their mean request arrival rates, respectively. For each CPU core, service a and b occupy $\rho_a = \lambda_a/(np\mu_{ua})$ and $\rho_b = \lambda_b/(np\mu_{ub})$ of CPU power. Therefore, for each CPU core, $1-\rho_b$ and $1-\rho_a$ are the available portion of CPU for service a and b, respectively. Since each service cannot use 100% of CPU power, their service rate are reduced to $p\mu_{ua}/(1-\rho_b)$ and $p\mu_{ub}/(1-\rho_a)$. These reduced service rates are applied to Equation 1 when calculating service instances' latency. When $\rho_a + \rho_b > 1$, i.e., CPU usage exceeds 100%, service rates are reduced so that $\rho_a + \rho_b = 1$ is hold.

Figure 4 illustrates an example. Instances of service a and b are deployed on a deployment plan with $n = 1$ and $p = 1$. λ_a is 3, μ_{ua} is 15, λ_b is 15 and μ_{ub} is 32. The CPU usage of a and b are $3/15 = 0.20$ and $15/32 = 0.47$, respectively. Since the CPU usage does not exceed the capacity ($0.20 + 0.47 < 1$), throughput of a and b are the same as λ_a and λ_b. (All incoming requests are processed.) The ratio of CPU power available for a is $1 - 0.47 = 0.53$, therefore a's mean service rate on this deployment plan is $0.54 \times 15 = 8.1$. The mean service rate of b is 25.6 as well. Then, the distribution of their latency is calculated based on their throughput and Equation 1.

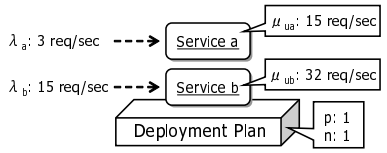

Figure 4: Multiple Service Instances

To illustrate the validity of this model Figure 5 to 7 compare latency (cumulative frequency) of a testbed system and theoretical estimates. A testbed is an HTTP server that has threads as the same number of CPU cores (e.g., two threads when running on a two core CPU) and request arrival times follow a Poisson process. In this experiment, what developers know is the only service rate of two services A and B on a 1.66 GHz CPU, i.e., 9.17 and 6.25 requests per second, respectively. Figure 5 compares latency of service A running on a CPU consisting of two 1.66 GHz cores under various request rates and theoretical estimates. (Dotted lines are measured latency and solid lines are theoretical estimates.) Figure 6 compares latency of service A running on a 2.2 GHz CPU and theoretical estimates. Figure 7 illustrates the case when service A and B are deployed on a deployment plan with a CPU consists of two 2.2 GHz cores. Service A and B receive requests at 12 and 4 requests per second, respectively. As these result illustrate $M/D/N$ queues approximate the latency of services under any CPU configurations and request arrival rates well.

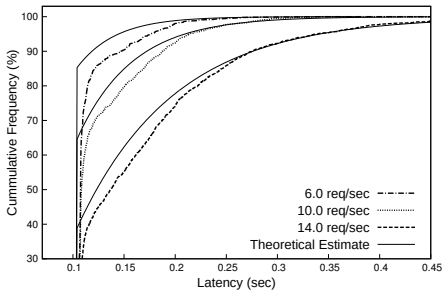

Figure 5: Latency on two 1.66 GHz cores

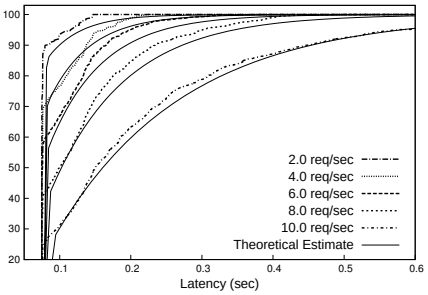

Figure 6: Latency on a 2.2 GHz CPU

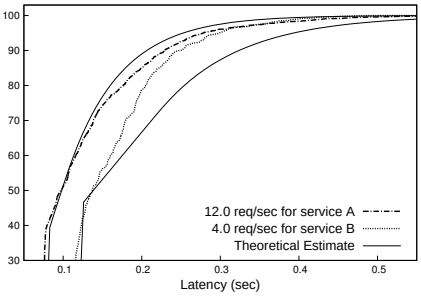

Figure 7: Latency on two 2.2 GHz cores

2.2 End-to-End QoS Aggregation

In order to examine whether a deployment configuration can achieve the end-to-end required throughput defined in a SLA, E^3/Q first distribute the required throughput to each service instance by leveraging a load balancing algorithm. For example, in Figure 2, when the end-to-end required throughput is 1,500 requests per second, a round robin algorithm distributes 500 requests to each instance of Service 1. Each instance of Service 2, 3 and 4 receives 750, 750 and 1500 requests per second, respectively. Then, E^3/Q examines each deployment plan whether its CPU usage exceeds 100% and calculates each service instances' throughput and latency distribution. The throughput of a service is determined as the summation of service instances' throughput. (e.g., service A's throughput is the summation of the throughput of all service A's instances.)

Then, the end-to-end throughput of a deployment plan is determined by applying QoS aggregate functions (Table 1) according to a workflow. For example, in Figure 1, the throughput of the part where Service 2 and Service 3 are connected in parallel is the minimum throughput of Service 2 or Service 3. Then, as the end-to-end throughput, the minimum throughput of Service 1, the parallel part and Service 4 is selected.

In order to obtain the probability distribution of the end-to-end latency E^3/Q employs Monte Carlo method since a simple aggregation (e.g., summation of each service's average latency) cannot reveal the probability distribution and lead to a too *pessimistic* estimation [5]. E^3/Q simulates the end-to-end latency of one request by (1) selecting services to execute (a path in a workflow) according to branching probabilities (all services are executed if a workflow has no branches.), (2) for each service selecting one of instances according to their throughput (an instance with larger throughput has higher change to be selected), (3) determining each instance's latency (a certain value) according to the probability distribution, and (4) aggregating the latency by applying aggregate functions in Table 1. By repeating this process many times, i.e., simulating many requests to an application, E^3/Q approximates the probability distribution of the end-to-end latency.

Since E^3/Q obtains the probability distribution of the end-to-end latency, it allows developers to define SLAs in a probabilistic manner, e.g., using best-case, worst-case, worst of the best 95%-case, average and the most frequent latency as QoS objectives. It also allows for using multiple types of latency in a SLA, e.g., a SLA can define both average and worst-case latency as QoS objectives.

Table 1: Aggregate Functions

QoS Attribute	Sequence	Branch	Parallel
Throughtput (T)	$\min_{s \in services} T_s$	$\sum_{b \in branches} T_s$	$\min_{s \in sequences} T_s$
Latency (L)	$\sum_{s \in services} L_s$	N/A	$\max_{s \in sequences} L_s$

3 Multiobjective Optimization of Service Deployment with E^3/Q

As a GA, E^3/Q maintains a population of individuals, each of which represents a service deployment configuration for each user category and encodes it as genes. E^3/Q evolves and optimizes the individuals in generations by repeatedly applying genetic operations to them. Currently, E^3/Q assumes three user categories: platinum, gold and silver users.

Figure 8 shows an example individual. An individual consists of three sets of genes, each of which represents a deployment configuration for each user category. A deployment configuration consists of deployment plans and service instances. An example in Figure 8 assumes that four types of deployment plans are available and three services are defined in a workflow. A deployment plan is encoded as a set of four genes; the first gene indicates the type of the deployment plan (i.e., 0 to 3 represents the index of the type) and the second to fourth genes indicate whether an instance of a certain service is deployed on it. (i.e., 1 indicates that an instance is deployed.) Therefore, the first four genes in the example, i.e., `2011`, represents a deployment plan of the third type that instances of the second and third services are deployed on. Since a deployment configuration can have arbitrary number of deployment plans, the number of genes varies depending on the number of deployment plans.

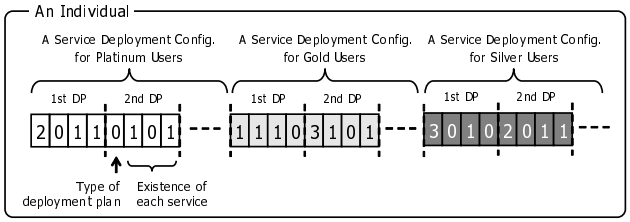

Figure 8: An Example Individual and Optimization Objectives

3.1 Genetic Operations in E^3/Q

Since the number of genes in an individual varies in E^3/Q, crossover and mutation, i.e., two most important genetic operations, must be designed to deal with it. A crossover operation in E^3/Q performs one-point crossover on genes for each user category. When it performs a crossover on two individuals, the crossover operation first picks a set of genes for platinum users from both two individuals, selects a crossover point on each genes and performs a crossover (Figure 9). A crossover point is randomly selected from points dividing deployment plans. For example, in Figure 8, a crossover point must be between $4i$-th and $4i+1$-th genes, e.g., 4th and 5th or 8th and 9th genes, since a deployment plan is encoded as a set of four genes. E^3/Q's crossover operation performs crossover on genes for gold users and silver users as well.

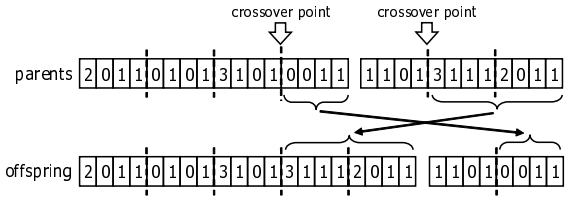

Figure 9: Crossover Operation in E^3/Q

A mutation operation in E^3/Q is designed to change the value of genes and the number of genes in an individual. First, in order to provide an opportunity to add a new deployment plan to a deployment configuration, the mutation operation adds an *empty* deployment plan, i.e., a deployment plan that no service instances run on it, before mutating genes (Figure 10). (The type of a deployment plan is randomly selected.) Mutation occurs on genes with the mutation rate of $1/n$ where n is the number of genes in an individual. When a mutation occurs on a gene used for specifying the type of a deployment plan, its value is randomly altered to represent another type of deployment plan. When a mutation occurs on a gene used for indicating the existence of a service instance, its value is changed from zero to one or one to zero. (i.e., non-existent to existent or existent to non-existent.) Therefore, mutation may turn a newly added empty deployment plan into non-empty and may turn existing non-empty deployment plans into empty. After that, the mutation operation examines each deployment plan and removes empty deployment plans from a deployment configuration. This way, the number of genes (the number of deployment plans) in an individual may change.

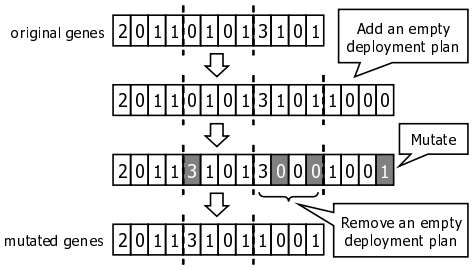

Figure 10: Mutation Operation in E^3/Q

3.2 Optimization Process in E^3/Q

Listing 1 shows the optimization process in E^3/Q. P^g denotes a population at the generation g. At each generation, two parents, p_α and p_β, are selected with binary tournaments. They reproduce two offspring by performing a crossover operation. Then, the offspring's genes are mutated. This reproduction process is repeated until the number of offspring (Q^g) reaches u. Of a union of P^g and Q^g, E^3/Q selects the top u individuals with respect to their fitness values. A fitness value indicates a quality (or goodness) of an individual; it is calculated with a fitness function in `AssignFitnessToIndividuals()`. E^3/Q repeats the above process for g_{max} times.

Listing 1: Evolution Process in E^3/Q

```
g ← 0
P^0 ← A population of randomly generated u individuals
repeat until g == g_max {
  AssignFitnessValuesToIndividuals(P^g)
  Q^g ← ∅
  repeat until |Q^g| == u {
    // Parent selection via binary tournament
    p_α, p_b ← Randomly selected two individuals from P^g
    p_α ← Either p_α or p_b with a higher fitness value
    p_α, p_b ← Randomly selected two individuals from P^g
    p_β ← Either p_α or p_b with higher fitness value

    // Reproduction via crossover
    q_1, q_2 ← Crossover(p_α, p_β)

    // Mutation on reproduced offspring
    q_1 ← Mutation(q_1)
    Add q_1 to Q^g if Q^g does not contain q_1.
    q_2 ← Mutation(q_2)
    Add q_2 to Q^g if Q^g does not contain q_2.
  }
  AssignFitnessValuesToIndividuals(P^g ∪ Q^g)
  P^{g+1} ← Top u of P^g ∪ Q^g in terms of their fitness values
  g ← g + 1
}

AssignFitnessValuesToIndividuals(P){
  DominationRanking(P)
  foreach p in P {
    if p is feasible
      // Fitness function for a feasible individual
      f ← p's domination value ×
          p's distance from the worst point ×
          p's sparsity
```

```
    else
      // Fitness function for an infeasible individual
      f ← 0 - p's SLA violation / p's domination value

    p's fitness value ← f
  }
}
```

E^3/Q is designed to seek individuals that satisfy given SLAs and exhibit the optimal trade-offs among QoS objectives in the SLAs. In order to fulfill the both requirements, E^3/Q distinguishes *feasible* individuals, which satisfy SLAs, and *infeasible* individuals, which do not. E^3/Q uses two different fitness functions for feasible and infeasible individuals. (See `AssignFitnessToIndividuals()`.) The fitness function for feasible individuals is designed to encourage them to improve their QoS values in all objectives and maintain diversity in their QoS values. The fitness function for infeasible individuals is designed to encourage them to reduce SLA violations and turn into feasible individuals. Feasible individuals have positive fitness values, while infeasible ones have negative fitness values. E^3/Q considers higher-fitness individuals as higher quality (or better). Therefore, feasible individuals have higher chances, than infeasible ones, to be selected as parents for reproduction.

3.3 Domination Ranking

In the design of fitness functions, E^3/Q employs the notion of *domination ranking* [6]. An individual i is said to *dominate* an individual j if any of the following conditions are hold.

1. Individual i is feasible and j is not.
2. Both i and j are feasible, and i outperforms j in terms of their QoS values.
3. Both i and j are infeasible, and i outperforms j in terms of their SLA violations.

In the second condition, an individual i is said to *outperform* an individual j if i's QoS values are better than, or equal to, j's in all QoS objectives, and i's QoS values are better than j's in at least one QoS objective. In the third condition, an individual i is said to *outperform* an individual j if i's SLA violations are lower than, or equal to, j's in all violated QoS attributes in SLAs, and i's SLA violations are lower than j's in at least one of violated QoS attributes. A SLA violation is measured as an absolute difference between an actual QoS measure and a required QoS level in a SLA.

Figure 11 shows an example of the second condition in domination ranking. It examines domination relations among six feasible individuals in terms of their QoS values. In this example, individuals A, B and C are at the first rank. In other words, they are *non-dominated*. A, B and C do not dominate with each other because each of them cannot dominate the others. Since individuals D and E are dominated by the top rank individuals, they are at the second rank. Similarly, individual F is at the third rank. A domination rank is assigned to an individual based on the number of individuals that dominate it.

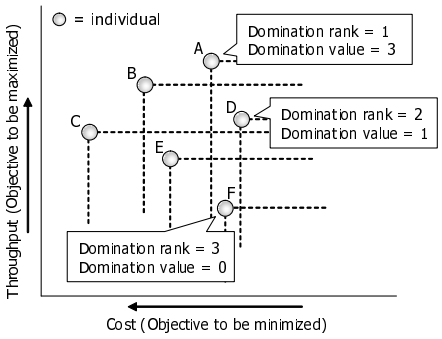

Figure 11: An Example Domination Ranking

3.4 Fitness Function for Feasible Individuals

For each feasible individual, E^3/Q assigns a fitness value based on its *domination value*, *distance from the worst point* and *sparsity*. (See `AssignFitnessToIndividuals()` in Listing 1.) These values contribute to increase a fitness value. An individual's domination value indicates the number of the other individuals in lower domination ranks than the individual in question. Therefore, non-dominated individuals have the highest domination values (Figure 11).

It is known that most individuals tend to become non-dominated when a problem has more than three objectives, and it weakens the pressure to evolve individuals since most solutions are in the same domination rank [7]. In order to avoid this issue, E^3/Q-MOGA uses the distance from the worst point, i.e., a point consists of the worst objective values in a population, to measure the excellence of individuals as well as domination values. For each feasible individual, E^3/Q-MOGA calculates the Manhattan distance from the worst point. (In this calculation, QoS values are normalized in case QoS objectives have different scales.) Manhattan distance is used because it becomes larger, compared with the other p-norm distances ($p \geq 2$) including Euclidian distance ($p = 2$), when individuals' objective values are balanced.

Sparsity represents diversity of individuals; how individuals spread uniformly in the objective space. Maintaining the diversity of individuals is an important consideration in E^3/Q-MOGA to reveal wider variety of trade-offs in objectives. For each individual, E^3/Q calculates the Manhattan distance to the closest neighbor individual in the objective space and determines the distance as its sparsity. (In this calculation, QoS values are normalized in case QoS objectives have different scales.) Manhattan distance is used be-

cause it becomes larger when individuals are uniformly distributed over all objectives. E^3/Q favors diversity of individuals because diverse individuals can reveal a wide range of trade-offs among QoS objectives.

3.5 Fitness Function for Infeasible Individuals

For each infeasible individual, E^3/Q assigns a fitness value based on its *total SLA violation* and domination value. (See `AssignFitnessToIndividuals()` in Listing 1.) The total SLA violation is calculated as the sum of violations against QoS requirements in SLAs. (In this calculation, violation values are normalized in case QoS requirements have different scales.) The total SLA violation contributes to decrease a fitness value, while a domination value contributes to increase it.

4 Simulation Evaluation

This section evaluates E^3/Q through a simulation study. This simulation study simulates a workflow shown in Figure 1. When running on a deployment plan with one 1.0GHz CPU, Service 1, 2, 3 and 4 process 28, 25, 23 and 20 req/sec, respectively. Table 2 are available deployment plans.

Table 2: Deployment Plans

Name	CPU Core Speed (GHz)	# of Cores	Cost ($)
High	1.0	4	50
Mid	1.2	2	30
Low	1.5	1	10

Table 3 shows the SLAs used in this simulation study. SLAs define each user category's throughput, worst of the best 95%-case latency (upper bound) and cost (upper bound). This simulation study assumes that 25 of platinum, 90 of gold and 750 of silver users access to an application on the average and allows uses in each category to send 2, 1 and 0.2 requests per second. Therefore, required throughput of platinum, gold and silver users is 50, 90 and 150 req/sec, respectively. Also, platinum and gold users have the worst case latency while they have no cost (or budget) limits. Silver users have a cost limit. In addition, there is a limit on the total costs incurred by all of three user categories. E^3/Q uses the population size (u) of 100 and the maximum generation (g_{max}) of 500.

Table 3: Service Level Agreements (SLAs)

User Category	SLAs			
	Throughput (req/sec)	95% Latency (sec)	Cost ($)	Total Cost ($)
Platinum	50	0.5	N/A	2,000
Gold	90	1.0	N/A	
Silver	150	N/A	400	

This problem has approximately 3.7×10^{49} of the search space. A deployment plan is represented as one of $D{\cdot}(2^S-1)$ combinations of genes where D is the number of deployment plan types and S is the number of services in a workflow. When a deployment configuration has M deployment plans, there are ${}_{D\cdot(2^S-1)}H_M = (M + D \cdot (2^S - 1) - 1)!/(M! \cdot (D \cdot (2^S - 1)! - 1))$ combinations of genes. Therefore, the SSDO problem has a search space of $\Sigma_{i=1}^{N}(i + D \cdot (2^S - 1) - 1)!/(i! \cdot (D \cdot (2^S - 1) - 1)!)$ where N is the maximum number of deployment plans in a deployment configuration. In this simulation study, D is three, S is four and N is 200 since the maximum total cost is $2,000 and the most inexpensive deployment plan is $10. Since E^3/Q is configured to run for 500 generations with 100 individuals, it examines 50,100 individuals in total, which is a very limited (only 2.2×10^{-44}%) regions in the entire search space.

Figure 12 to 19 show average objective values that feasible individuals yield. Results are obtained through 50 independent GA runs. Throughput is not shown here since feasible solutions' throughput is the same as that in SLAs. (e.g., Throughput of all feasible solutions is exactly 50 req/sec for platinum users.)

Figure 12, 13 and 14 show the maximum, average and minimum latency that individuals yield for platinum, gold and silver users. Individuals successfully evolve and satisfy SLAs for platinum and gold users. Although silver users do not have a SLA for latency, E^3/Q lowers latency for them with the other SLAs satisfied. Moreover, the maximum latency of each user category increases over generations, i.e., individuals evolve and reveal trade-offs in latency. For example, application developers can select balanced one or one with extremely high latency for silver users for optimizing other objectives such as costs.

Figure 15, 16 and 17 show the maximum, average and minimum cost that individuals incur for platinum, gold and silver users. Figure 18 shows the total cost. Individuals successfully evolve and satisfy SLAs for silver users. Although platinum and gold users have no SLAs for costs, E^3/Q lowers costs for them with the other SLAs satisfied. As well as latency, individuals evolve to reveal trade-offs in cost over generations.

Figure 19 shows the the number of (1) feasible individuals and (2) feasible and non-dominated individuals over generations. Although all individuals become feasible and non-dominated at around 100th generation, individuals keep evolving to obtain better objective values and reveal wider trade-offs sine the fitness function in E^3/Q considers distance from the worst point as well as domination values (see Section 3.4).

Figure 20 shows results of another simulation study that investigates the efficiency of E^3/Q. This simulation study sets up several deployment configurations whose search space is 700 to 3.9×10^4 and obtains a set of true optimal

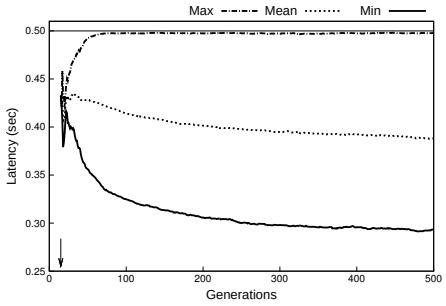

Figure 12: Latency of Platinum Users

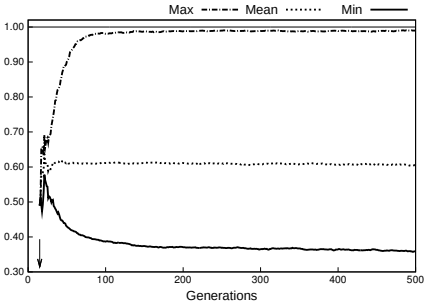

Figure 13: Latency of Gold Users

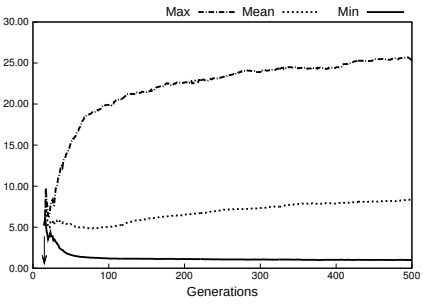

Figure 14: Latency of Silver Users

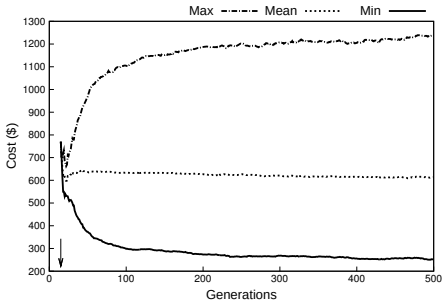

Figure 15: Cost of Platinum Users

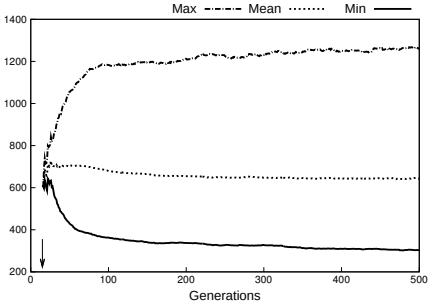

Figure 16: Cost of Gold Users

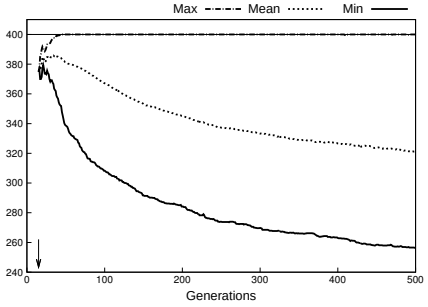

Figure 17: Cost of Silver Users

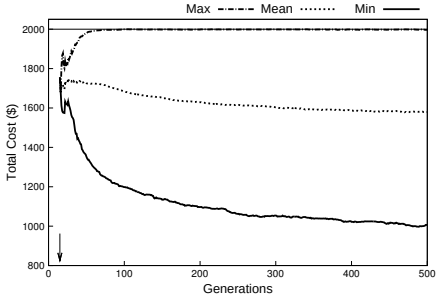

Figure 18: Total Cost

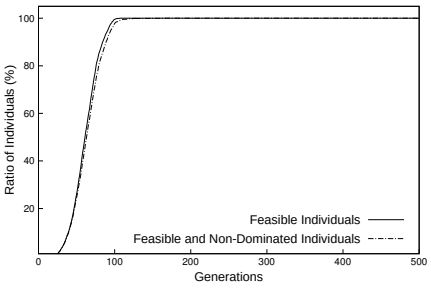

Figure 19: Ratio of Feasible Solutions

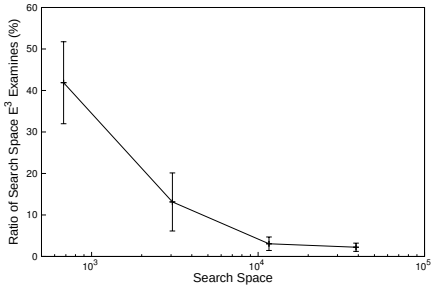

Figure 20: Ratio of Search Space Examined

solutions by leveraging brute force searches. Then, E^3/Q searches optimal solutions in the same set of deployment configurations. Figure 20 shows the ratio of search space that E^3/Q examines to find true optimal solutions. (Average and standard deviation of independent ten runs.) The ratio decreases as the search space increases, and E^3/Q can find optimal solutions in a very short time even in problems with huge search space.

5 Related Work

This paper describes a set of extensions to the authors' prior work [8]. Extensions includes (1) a performance estimation using queuing theory, (2) a revision of service deployment and QoS models, (3) a support of probabilistic SLAs, and (4) a revision of E^3/Q-MOGA. By leveraging a queuing theory based performance estimation and QoS aggregation through Mote Carlo method, this paper allows for defining SLAs in a probabilistic manner. Also, E^3/Q's fitness functions are revised to represent the excellence of solutions more properly.

Various research efforts have investigated the SSDO problem. However, most of them do not support assumptions that the current clouds make. For example, they do not consider binding multiple service instances to a service. No existing work considers differentiating SLAs for different user categories of an application. E^3/Q is the first attempt to investigate the SSDO problem in the context of cloud computing.

Currently, linear programming is a major method to solve the SSDO problem (e.g., [9, 10]). However, it is not designed to seek trade-offs among conflicting optimization objectives. It also has a scalability issue; its computational cost increases exponentially as a search space grows [1]. In practice, the SSDO problem has a huge search space as dis-

cussed in Section 4. Therefore, linear programming does not work well in the SSDO problem for large-scale applications.

In order to solve large-scale SSDO problems, it is required to use heuristic methods such as GAs [1, 11]. In general, GAs scale better than linear programming. However, it is always non-trivial to manually tune weight parameters in a fitness function of a classical GA. (A classical GA has a fitness function as a weighted sum of objective values.) Also, similar to linear programming, classical GAs do not seek the optimal trade-offs among conflicting objectives. Multiobjective GAs avoids the above issues in classical GAs. They seek the optimal trade-off (or Pareto-optimal) solutions, and have no weight parameters in their fitness functions thanks to domination ranking. A limited number of research efforts have investigated multiobjective GAs for the SSDO problem (e.g., [12, 13]). However, none of them consider SLAs as E^3/Q does.

A number of research leverage queuing theory to estimate the performance of distributed systems [2, 3]. E^3/Q is the first attempt to apply queuing theory to the SSDO problem. Also, most of research work use queuing theory to estimate only average latency and do not allow for considering the probability distribution. [5] leverages Monte Carlo method to aggregation QoS and obtain the probability distribution. However, it uses the probability distributions to estimate the performance of service oriented applications rather than optimize them as E^3/Q does.

Several research have investigated methods to aggregate SLAs in service oriented applications [14]. They propose functions to aggregate a set of SLAs into end-to-end SLAs when an application consists of multiple services that have their own SLAs. E^3/Q assumes each service in an application has no SLAs and QoS aggregate functions are used to aggregate service instances' QoS measures to investigate whether aggregated QoS measures satisfies application's end-to-end SLAs. Therefore, aggregation of SLAs is out of the scope of current E^3/Q.

6 Conclusion

This paper proposes and evaluates E^3/Q, which is a multiobjective GA to solve the SSDO problem for service-oriented cloud applications. Simulation results demonstrate that E^3/Q efficiently obtains quality deployment configurations that satisfy given SLAs. As future work, empirical evaluations are planned to use E^3/Q with several clouds and provide additional performance implications.

7 Acknowledgement

This work is supported in part by OGIS International, Inc.

References

[1] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. An Approach for QoS-aware Service Composition based on Genetic Algorithms. In *ACM International Conference on Genetic and Evolutionary Computation Conference*. ACM Press, June 2005.

[2] C. Stewart, T. kelly, and A. Zhang. Exploiting Nonstationarity for Performance Prediction. In *ACM European Conference on Computer Systems*, Mar 2007.

[3] Y. Liu, I. Gorton, and L. Zhu. Performance Prediction of Service-Oriented Applications based on an Enterprise Service Bus. In *IEEE Int'l Computer Software and Applications Conference*, July 2007.

[4] W. C. Chan and Y. B. Lin. Waiting time distribution for the M/M/m queue. *IEE Proceedings Communications*, 150(3):159–162, 2003.

[5] P. QoS and S. C. f Transaction-Based Web Services Orchestrations. Model Driven Development for Business Performance Management. *IEEE Transactions on Services Computing*, 1(4):187–200, 2006.

[6] N. Srinivas and K. Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.

[7] H. Ishibuchi, N. Tsukamoto, Y. Hitotsuyanagi, and Y. Nojima. Effectiveness of scalability improvement attempts on the performance of NSGA-II for many-objective problems. In *ACM International Conference on Genetic and Evolutionary Computation Conference*, July 2008.

[8] H. Wada, P. Champrasert, J. Suzuki, and K. Oba. Multiobjective Optimization of SLA-aware Service Composition. In *IEEE Workshop on Methodologies for Non-functional Properties in Services Computing*, July 2008.

[9] T. Yu, Y. Zhang, and K. J. Lin. Efficient Algorithms for Web Services Selection with end-to-end QoS Constraints. *ACM Transactions on the Web*, 1(1), Dec 2007.

[10] D. Ardagna and B. Pernici. Adaptive Service Composition in Flexible Processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, June 2007.

[11] M. C. Jaeger and G. Mühl. QoS-based Selection of Services: The Implementation of a Genetic Algorithm. In *Workshop on Service-Oriented Architectures and Service-Oriented Computing*, March 2007.

[12] D. B. Claro, P. Albers, and J. Hao. Selecting Web Services for Optimal Composition. In *IEEE International Workshop on Semantic and Dynamic Web Processes*, July 2005.

[13] H. A. Taboada, J. F. Espiritu, and D. W. Coit. MOMS-GA: A Multi-Objective Multi-State Genetic Algorithm for System Reliability Optimization Design Problems. *IEEE Transactions on Reliability*, 57(1):182–191, March 2008.

[14] T. Unger, F. Leymann, S. Mauchart, and T. Scheibler. Aggregation of Service Level Agreements in the Context of Business Processes. In *IEEE Int'l Enterprise Distributed Object Computing Conference*, Sept 2008.