An Evolutionary Algorithm for Performance Optimization at Software Architecture Level

Xin Du
Faculty of software
Fujian Normal University
Fuzhou, China
School of Computer Science
The University of Birmingham
Birmingham, UK
duxy@cs.bham.ac.uk

Youcong Ni
Faculty of software
Fujian Normal University
Fuzhou, China
youcongni@foxmail.com

Peng Ye
College of Mathematics and Computer
Wuhan Textile University
Wuhan, China

Xin Yao
School of Computer Science
The University of Birmingham
Birmingham, UK

Leandro L. Minku
School of Computer Science
The University of Birmingham
Birmingham, UK

Ruliang Xiao
Faculty of software
Fujian Normal University
Fuzhou, China

Abstract— Architecture-based software performance optimization can not only significantly save time but also reduce cost. A few rule-based performance optimization approaches at software architecture (SA) level have been proposed in recent years. However, in these approaches, the number of rules being used and the order of application of each rule are uncertain in the optimization process and these uncertainties have not been fully considered so far. As a result, the search space for performance improvement is limited, possibly excluding optimal solutions. Aiming to solve this problem, we propose an evolutionary algorithm for rule-based performance optimization at SA level named EA4PO. First, the rule-based software performance optimization at SA level is abstracted into a mathematical model called RPOM. RPOM can precisely characterize the mathematical relation between the usage of rules and the optimal solution in the performance improvement space. Then, a framework named RSEF is designed to support the execution of rule sequences. Based on RPOM and RSEF, EA4PO is proposed to find the optimal performance improvement solution. In EA4PO, an adaptive mutation operator is designed to guide the search direction by fully considering heuristic information of rule usage during the evolution. Finally, the effectiveness of EA4PO is validated by comparing EA4PO with a typical rule-based approach. The results show that EA4PO can explore a relatively larger space and get better solutions.

Keywords—performance analysis; performance optimization algorithm; evolutionary algorithm; software architecture; search-based software engineering; rule

I. INTRODUCTION

Performance is an important quality attribute of software systems and is a vital factor to determine success or failure of a system. Architecture-based software performance diagnosis and improvement can find the problems (e.g. high resource utilization, long response time and low throughput) and provide the solutions to mitigate the negative effects of them at the early stage of the software life cycle. Architecture-based software performance optimization can not only significantly save time but also reduce cost.

Architecture-based software performance optimization has been a highlight topic among academia and industry in the field of software engineering. There have been a few models and tools [1, 2] for architecture-based software performance evaluation, such as Queuing Network (QN) [3], Layered QN (LQN) [4, 5], Stochastic Petri Nets (SPN) [6], Stochastic Process Algebras (SPA) [7] and Markov process [8]. They can strongly support architecture-based software performance diagnosis and improvement. However, along with the increasing size and complexity of software systems, the factors impacting system performance grow and the space of architecture-based software performance improvement also enlarges. More importantly, this space is intrinsically discontinuous because various constraints and limitations need to be satisfied in the process of performance optimization. It is still a difficult problem for software performance optimization to find the optimal or near-optimal performance improvement solution in a huge and discrete performance improvement space. To solve this problem, some researchers have presented metaheuristic-based approaches [9, 10] and rule-based approaches [11-14] in recent years. At present, metaheuristic-based approaches can only explore a few of architectural parameters, such as component allocation, hardware configuration, and component selection. Moreover, the majorities of metaheuristic-based approaches do not consider how to apply architecture-based software performance improvement knowledge in the evolutionary optimization process. The disadvantages of metaheuristic-based approaches can be partly avoided in rule-based approaches.

In the typical rule-based approaches including Xu's [11], McGregor's [12] and Cortellessa's [13, 15, 16], the architecture-based software performance improvement

978-1-4799-7492-4/15/$31.00

knowledge from anti-patterns [17, 18] or the specified application domains [19] can be formally described in the form of rules. These rules usually cover many parameters from various viewpoint of *SA*, such as structure, behavior and deployment. Each rule contains two parts of the condition and action. The condition of rule is responsible for diagnosing the performance problem. Meanwhile the actions of rules describe potential improvements for the discovered performance problems. In addition, engines or frameworks have been developed to automatically [11, 12] or semi-automatically [15] apply these rules to find the performance problems and eliminate them by predefined improvement solutions.

In rule-based approaches, the number and order of each rule usage are two key factors that affect the results of performance optimization. For example, consider that there are two performance improvement rules *r1* and *r2*. Rule *r1* can be used to solve the bottleneck problem of processor *P* by increasing *P*'s multiplicity, while rule *r2* can diagnose bottleneck problem of component *C* and alleviate it by raising *C*'s multiplicity. Suppose that component *C1* is deployed on processor *P1*, and that there are bottleneck problems in both *C1* and *P1*. It is clear that these problems can be found in the initial *SA* by the rules *r1* and *r2*, respectively. However, the optimization results depend on the count and order of the rules *r1* and *r2* usage during performance optimization.

If the bottleneck of processor *P1* can not be eliminated after rule *r1* is applied once and the *P1*'s multiplicity is increased with a specified increment, rule *r1* can also be applied several times provided that *P1*'s multiplicity is less than its pre-defined maximum multiplicity. After rule *r1* is repeatedly applied several times, the bottleneck problem of *C1* is likely to disappear because the bottleneck of *P1* on which *C1* is deployed has been solved. Then, there is no further improvement when rule *r2* is applied.

In another instance where rule *r2* is applied before rule *r1*, the bottleneck problem of processor *P1* will become more serious and rule *r1* may need to be applied more times than expected. Thus, the optimization results will vary according to the count and order of each rule usage. Unfortunately, how to use these rules is uncertain

Due to the existence of these uncertainties, in practice, the improvement space is enormous. For example, Cortellessa and Trubiani et al presented 12 performance improvement rules [15] for 12 architecture-based performance anti-patterns [17]. So, the size of the performance improvement space is determined by permutation and combination of 12 different performance improvement rules. This space contains 12^{12} (nearly 9,000 trillion) possible solutions even if the repetitive usage of each rule in a solution is not considered. To find optimal performance improvement solution, the existing rule-based approaches may only explore a limited region of the improvement space because they do not fully take the uncertainty of the number and order of each rule usage into account. Obviously, the limitation of the search space easily leads to the fact that the optimal performance improvement solution is hard to find, possibly even unreachable by the algorithm. Consequently, the quality of performance optimization is likely to be suboptimal.

Aiming at solving the problems above, first, a rule-based performance optimization model named RPOM is presented to depict precisely the mathematical relation between the rules usage and the optimal solution in the performance improvement space. Then, a framework called RSEF is designed to support the rule sequence execution. Based on RPOM and RSEF, EA4PO is proposed to find the optimal solution. In EA4PO, an adaptive mutation operator with learning tactics is introduced to improve convergence rate. Finally, experiments are done to validate the effectiveness of EA4PO by comparing our work with Xu's. The results indicate that EA4PO can explore a relatively larger space and get better solutions than Xu's algorithm.

The rest of this paper is organized as follows. Section II introduces the RPOM model. The RSEF framework and EA4PO are proposed in Sections III and IV. Section V presents a case study. Finally, Section VI concludes this paper.

II. THE RPOM MODEL

Rule-based performance optimization model (RPOM) is defined as follows:

The performance improvement rules are sequentially numbered from 1 to *n*. Let a sequence of rule numbers $X =< x_1, \cdots, x_k, \cdots, x_l >$ be a solution for the problem of rule-based performance optimization at *SA* level, where *l* is the length of *X*. Let u_i and $h_i(X)$ represent the maximal possible occurrence times and actual occurrence times of rule *i* in *X* respectively.

The constraints for *l*, the value range of each element in *X* and $h_i(X)$ are defined as formulas (1), (2) and (3), respectively, where *i*, x_k , *k*, *l* and *n* represent natural numbers.

$$l \le \sum_{i=1}^{n} u_i \tag{1}$$

$$1 \le x_k \le n \ \wedge \ 1 \le k \le l \tag{2}$$

$$h_i(X) \le u_i \text{, where } 1 \le i \le n \tag{3}$$

In the following definition of functions, let *SA* and SA_0 be the software architecture and the initial software architecture, respectively.

The function $imp(q, SA)$ can determine whether the performance has been improved by applying rule *q* to *SA*.

The function *imp* is defined as formula (4).

$$imp(q, SA) = \begin{cases} 0, \ if \ \text{the performance has not} \\ \quad \text{been improved by applying} \\ \quad \text{the rule numbered } q \text{ to } SA, \ q \in N \wedge 1 \le q \le n \\ 1, \ \text{otherwise} \end{cases} \tag{4}$$

The function $t(X, SA)$ can acquire the improved software architecture by applying sequentially the rules indicated by *X to SA*. The function *t* is defined as formulas (5) and (6), where *ISA* is the improved software architecture after applying x_k to *SA*.

$$t(< x_k >, SA) = \begin{cases} SA, & imp(x_k, SA) = 0 \\ ISA, & imp(x_k, SA) = 1 \end{cases}, 1 \le k \le n \quad (5)$$

$$t(X, SA) = t(< x_2, \cdots, x_l >, t(< x_1 >, SA)) \quad (6)$$

The function *impRulCount*(*X*, SA_0) is used to compute the count of rules usage with improvement effect in the process of applying sequentially the rules indicated by *X* to SA_0. The definition of *impRulCount*(*X*, SA_0) is as formula (7). In formula (7), $V_k =< x_1, \cdots, x_{k-1} >$.

$$impRuleCount(X, SA_0) = imp(x_1, SA_0) + \sum_{k=2}^{l} imp(x_k, t(V_k, SA_0)) \quad (7)$$

The function *r* (*SA*) returns the response time of the system specified by *SA*.

Function *g*(*X*) is defined as objective function of performance optimization. The definition of the *g*(*X*) is shown as formula (8).

$$g(X) = (r(SA_0) - r(t(X, SA_0))) - k * impRuleCount(X, SA_0) \ , 0 < k < 1 \quad (8)$$

Overall, *g(X)* is the difference between two terms. The first term represents the extent of performance improvement, while the second term is product of a weight factor and the count of rules usage with improvement effect. The larger the return value of *g*(*X*), the better the corresponding solution *X*. Here, a smaller number of rules may be better because it would lead to less modifications in the architecture.

The rule-based software performance optimization at SA level can be formally described as: to find solution *X** which satisfies formula (9).

$$\underset{X \in \aleph}{Max}\{g(X)\} = g(X^*) \quad (9)$$

In formula (9), *X* satisfies the constraints defined by formulas (1), (2) and (3). $\aleph$ represents the solution space.

III. RSEF FRAMEWORK

The rule sequence execution framework (RSEF) is proposed to realize the functions *imp*, *t* and *r* and support the computation of the objective function *g* in RPOM based on existing rule engine. Its structure is shown in Fig. 1.

RSEF encompasses the control engine and the rule execution engine. Based on the two engines and the related data structure, the RSEF framework can support execution of the rule sequence during the performance optimization. The data structure and the execution process with respect to RSEF are defined as follows.

A. The definition of data structure

(1) The performance improvement rule

A performance improvement rule is stored as a file in the specific internal format. Each performance improvement rule is numbered by a unique integer and can be executed by the specified rule execution engine.

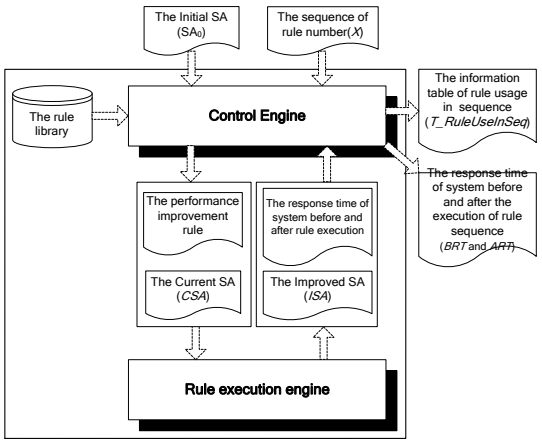

Fig. 1. The structure of RSEF framework

(2) The sequence of rule number *X*

A sequence of rule number *X* is defined as an integer array in which each element represents the rule number. It is a solution defined in RPOM.

(3) The rule library

A library stores all numbered performance improvement rules and each rule can be obtained from this library according to its number.

(4) The initial *SA*

The initial *SA* is developed by the architects and stored as a file in particular form. It is denoted as SA_0 in RPOM.

(5) The current *SA* and the improved *SA*

The current *SA* (*CSA*) is a file which is used as input of the rule execution engine, and the improved *SA (ISA)* is the output file which can be generated after the rule execution engine applies a performance improvement rule to the current *SA*. The *CSA* and *ISA* is input and output of a calculation step of the function *t* defined in RPOM.

(6)The response time of system before and after rule execution

They are two response times computed by rule execution engine based on the *CSA* and *ISA*, respectively. They are output of the function *r* defined in RPOM.

(7)The response time of system before and after the execution of rule sequence (*BRT* and *ART*)

They are two response times, and can be acquired based on SA_0 and the *SA* which can be obtained after the rule sequence is executed on SA_0. They are defined as $r(SA_0)$ and $r(t(X, SA_0))$ in RPOM, respectively.

(8)The information table of rules usages in the sequence

This table is called *T_RuleUseInSeq* whose fields are shown in Table I. The primary key of the *T_RuleUseInSeq* is composed of two fields of *loc* and *rulNum*. *T_RuleUseInSeq* can be used to compute values of the two functions *imp* and *impRuleCount* defined in RPOM.

TABLE I. THE FIELDS OF TABLE T_RULEUSEINSEQ

No.	Field	Description
1	*loc*	The index position of an element in a rule sequence *X*
2	*rulNum*	The rule number at the *loc* position in a rule sequence *X*
3	*isImp*	When the rule numbered *rulNum* in the *loc* position is applied to the current *SA*, if response time gets shorter, *isImp* is assigned 1. Otherwise it is 0.

B. The definition of execution process of rule sequence

The execution process of rule sequence is described by Algorithm 1 named *ARE*.

ALGORITHM 1 THE ARE ALGORITHM

Input: X, SA_0
Output: *BRT*, *ART*, *T_RuleUseInSeq*

1: $CSA \leftarrow SA_0$, $i \leftarrow 1$, $BRT \leftarrow -1$, $ART \leftarrow -1$ and let *T_RuleUseInSeq* be empty;
2: **while** $i <= |X|$ do
3: The rule execution engine runs after receiving *CSA* and the performance improvement rule numbered x_i;
4: $\Delta_{RT} \leftarrow BRT_i - ART_i$;
5: **if** BRT= -1 **then**
6: $BRT = BRT_i$;
7: **end if**
8: $ART \leftarrow ART_i$
9: **if** $\Delta_{RT} > 0$ **then**
10: insert the record (i, x_i, 1) into the *T_RuleUseInSeq*;
11: **else**
12: insert the record (i, x_i, 0) into the *T_RuleUseInSeq*;
13: **end if**
14: **if** $\Delta_{RT} > 0$ **then**
15: Let *CSA* be the *SA* which returned by rule execution engine;
16: **end if**
17: $i \leftarrow i+1$;
18: **end while**
19: Output *BRT*, *ART* and *T_RuleUseInSeq*.

IV. EA4PO ALGORITHM

Based on RPOM model and RSEF framework, EA4PO algorithm is proposed to find the optimal improvement solution. EA4PO algorithm is elaborated as following.

A. Individual encoding

An individual $X' = < x'_1, x'_2, \cdots, x'_k, \cdots, x'_{l'} >$ is encoded as a sequence of rule number with fixed-length integer. The length l' of X' is defined as formula (10).

$$l' = \sum_{i=1}^{n} u_i \tag{10}$$

In formula (10), u_i represents maximal possible occurrence times of rule number i in X'. In order to ensure that each rule number in a sequence can comply with formulas (1)-(3) defined by RPOM and get the shorter optimal sequence of rule numbers, rule number 0, which indicates the do-nothing rule, is introduced into individual encoding. If the do-nothing rule is applied to any *SA*, the performance of system cannot be improved. Let u_0 represent the maximal possible occurrence times of the rule number 0 in X' and $u_0 = l'$.

Any rule number x'_k in X' satisfies formula (11).

$$x'_k \in N \wedge 0 \le x'_k \le n \quad (1 \le k \le l') \tag{11}$$

The actual occurrence times of rule number i in X' is denoted by $h_i(X')$ which satisfies formula (12).

$$h_i(X') \le u_i, 0 \le i \le n \tag{12}$$

B. Fitness function

The fitness function of individual X' is defined as formula (13). The larger the value of $fitness(X')$, the better the corresponding solution X'.

$$fitness(X') = (BRT - ART) - k \times impRuleCount(X, SA_0),\ 0 < k < 1 \tag{13}$$

In formula (13), X is a sequence of rule number which is generated by deleting rule number 0 in X'. It is same as the formula (8) in RPOM. *BRT* and *ART* represent the response time of system before and after the execution of rule sequence X respectively.

The algorithm ***solveFitness*** is designed to compute the fitness value of X'. In ***solveFitness*** algorithm, there is a statistical table of rule usage called *T_RuleUseInHis*, which is designed to record the information of rules usage during the evolution. The fields of the *T_RuleUseInHis* are shown in Table II. Algorithm 2 depicts the ***solveFitness*** in detail.

TABLE II. THE FIELDS OF TABLE *T_RULEUSEINHIS*

No.	Field	Description
1	*loc*	The index position of an element in a rule sequence *X*
2	*rulNum*	The rule number at the *loc* position in a rule sequence *X*
3	*preRulNum*	The rule number at the *loc*-1 position in a rule sequence *X*. And if *loc*=1, then let *preRulNum*= -1.
4	*impNum*	The usage count of rule numbered by *rulNum* with improvement effect under the condition that the two rule numbers *rulNum* and *preRulNum* are in the *loc* and *loc*-1 positions respectively
5	*totNum*	The usage count of rule numbered by *rulNum* under the condition that the two rule numbers *rulNum* and *preRulNum* are in the *loc* and *loc*-1 position respectively

ALGORITHM 2 THE SOLVEFITNESS ALGORITHM

Input: individual X', SA_0, *T_RuleUseInHis*
Output: the fitness value of X', *T_RuleUseInHis*

1: Get X by deleting all 0 in X';
2: Run the **ARE algorithm** in RSEF framework by inputting X and SA_0, and get *T_RuleUseInSeq*, *BRT* and *ART*;
3: Compute *impRulCount*(X,SA_0) according to *T_RuleUseInSeq* ;
4: Compute the fitness value of X' based on formula (13);
5: Run the **Updating algorithm** by inputting *T_RuleUseInSeq and* Update *T_RuleUseInHis*;

6: Output the fitness value of X' and update *T_RuleUseInHis.*

ALGORITHM 3 UPDATING ALGORITHM

Input: *T_RuleUseInSeq , T_RuleUseInHis*
Output: *T_RuleUseInHis*

1: Let $i \leftarrow 1$, $len \leftarrow$ the number of records in Table *T_RuleUseInSeq*;
2: **While** $i \leq$ len do
3: Take the i^{th} record (loc_i, $rulNum_i$, $isImp_i$) from the Table *T_RuleUseInSeq*;
4: Search the record and assign it to $record_j$ according to the condition $(loc = loc_i) \wedge (rulNum = rulNum_i) \wedge (preRulNum = x_{loc_i - 1})$ in Table *T_RuleUseInHis*. If loc_i=1, then *preRulNum*= -1;
5: **If** $isImp_i = 1 \wedge record_j \neq null$ **then**
6: Add 1 to value of the field *impNum* and *totNum* in $record_j$, respectively and update the *record* corresponding to $record_j$ in Table *T_RuleUseInHis*;
7: **end if**
8: **If** $isImp_i = 0 \wedge record_j \neq null$ **then**
9: Add 1 to value of the field *totNum* in $record_j$ and update the record corresponding to $record_j$ in Table *T_RuleUseInHis*;
10: **end if**
11: **If** $(isImp_i = 1) \wedge (record_j = null) \wedge (loc_i \neq 1)$ **then**
12: Insert the record $(loc_i, rulNum_i, x_{loc_i - 1}, 1, 1)$ into the Table *T_RuleUseInHis*;
13: **end if**
14: **If** $(isImp_i = 1) \wedge (record_j = null) \wedge (loc_i = 1)$ **then**
15: Insert the record $(loc_i, rulNum_i, -1, 1, 1)$ into the Table *T_RuleUseInHis*;
16: **end if**
17: **If** $(isImp_i = 0) \wedge (record_j = null) \wedge (loc_i \neq 1)$ **then**
18: Insert the record $(loc_i, rulNum_i, x_{loc_i - 1}, 0, 1)$ into the Table *T_RuleUseInHis*;
19: **end if**
20: **If** $(isImp_i = 0) \wedge (record_j = null) \wedge (loc_i = 1)$ **then**
21: Insert the record $(loc_i, rulNum_i, -1, 0, 1)$ into the Table *T_RuleUseInHis*;
22: **end if**
23: $i \leftarrow i+1$;
24: **end while**
25: Output *T_RuleUseInHis*

C. Crossover operator

One-point crossover with constraint checking and repairing is adopted. And it includes three computational steps: crossover, constraint checking and repairing. First, two intermediate individuals are generated by crossover operator. Then, the constraint checking is done on two intermediate individuals to verify whether each bit from the crossover position to the last position satisfies the constraint defined in formula (3). Last, repairing step will be done for those bits which disobey the constraint and assign 0 to them.

For example, there are two parent individuals $X'_1 =< 1,2,3,3,2,1,3,1 >$ and $X'_2 =< 2,1,2,3,3,1,1,3 >$. Each individual is composed of the rule numbers 1, 2, 3 and their maximal possible occurrence times are defined as u_1=3 , u_2=2 , u_3=3 . So, the length of each individual is 8. Suppose that the crossover point is randomly chosen as 5, the process of one-point crossover in EA4PO algorithm can be described as follows. First, two intermediate individuals $X'_{11} =< 1,2,3,3,3,1,1,3 >$ and $X'_{21} =< 2,1,2,3,2,1,3,1 >$ are generated by exchanging their corresponding bits from crossover position to the last position. Second, constraint checking is done on X'_{11} and X'_{21}. The 8^{th} bit in X'_{11} and the 5^{th} bit in X'_{21} disobey the constraints which are $h_3(X'_{11}) = 4 > u_3$ and $h_2(X'_{21}) = 3 > u_2$. Third, the two bits should be assigned as 0. As a result, two offspring individuals $X'_3 =< 1,2,3,3,3,1,1,0 >$ and $X'_4 =< 2,1,2,3,0,1,3,1 >$ are formed.

D. Adaptive mutation operator

The mutation operator has a great influence on the convergence rate of evolutionary algorithm [20]. Here, the adaptive mutation operator with learning tactics is introduced to guide the search direction of algorithm. Each bit mutates according to the conditional mutation probabilities that are drawn from the statistical table of rule usage *T_RuleUseInHis* during the evolution. The mutation operator in EA4PO algorithm is composed of three computational steps: mutate, constraint checking and repairing.

The conditional mutation probabilities $p(x'_j = k \mid x'_{j-1} = q)$ are computed based on the statistical table of rules usage *T_RuleUseInHis.* And the statistical information can be acquired by the functions $f_\Delta(j,k,m)$ and $O(j,k)$ described in the following.

The function $f_\Delta(j,k,m)$ aims to obtain the average improvement rate of the k^{th} rule under the condition that the rule number k and m are at j and j-1 positions respectively. The function $O(j,k)$ can be used to obtain all the rule numbers at j+1 position under the condition that rule number k is at j position.

On the basis of the functions $f_\Delta(j,k,m)$ and $O(j,k)$, the conditional mutation probability $p(x'_j = k \mid x'_{j-1} = q)$ can be defined as formula (14). Especially, let j (j>=1) represent mutation position and $x'_0 = -1$.

$$p(x'_j = k \mid x'_{j-1} = q) = \begin{cases} f_\Delta(1,k,-1) / \sum_{i \le n \wedge i \neq x'_j} f_\Delta(1,i,-1), & j = 1 \wedge k \neq x'_j \\ f_\Delta(j,k,q) / \sum_{i \in (O(j-1,q)/x'_j)} f_\Delta(j,i,q), & j > 1 \wedge k \neq x'_j \wedge O(j-1,q)/x'_j \neq \Phi \\ u_k / \sum_{1 \le i \le n \wedge i \neq x'_j} u_i, & j > 1 \wedge k \neq x'_j \wedge O(j-1,q)/x'_j = \Phi \end{cases} \quad (14)$$

In formula (14), the definition of $p(x'_j = k \mid x'_{j-1} = q)$ includes three cases. The first is mutation position $j = 1$. The second is that the mutation position $j > 1$ and the set of rule numbers returned by function $O(j-1,q)$ after removing the rule number k is not empty set Φ. The third is that the mutation position $j > 1$ and the set of rule numbers

returned by function $O(j-1,q)$ after removing the rule number k is empty set Φ .

E. The main steps of EA4PO algorithm

Let $X^{*\prime}$ represent the optimal individual in EA4PO, X^* represent the optimal sequence by deleting all 0 in $X^{*\prime}$.The main steps of EA4PO algorithm is given in Algorithm 4.

ALGORITHM 4 THE PROPOSED EA4PO ALGORITHM

Input： popsize n, crossover rate p_c, mutation rate p_m
Output：**the** optimal sequence X^*
1: Initialize evolutionary generation $t \leftarrow 0$;
2: Create an initial population $P(t)$ with n individuals;
3: Create an empty Table *T_RuleUseInHis*;
4: Compute the fitness value of each individual in $P(t)$ by using the algorithm ***solveFitness***;
5: Select the optimal individual from population $P(t)$ and assign it to $X^{*\prime}$
6: **While** stop criteria is not satisfied **do**
7: Generate a new (intermediate) population by defined one-point crossover operator based on population $P(t)$ with probability p_c ($0 < p_c < 1$) and denote it as $P_C(t)$;
8: Generate a new (intermediate) population by adaptive mutation operator based on population $P_C(t)$ with probability p_m ($0 < p_m < 1$) and denote it as $P_M(t)$;
9: Compute the fitness value of each individual in $P_M(t)$ by using the algorithm ***solveFitness***;
Select n individuals from population $P_M(t)$ and $P(t)$ according to elitist-based roulette select, then generate the next
10: population $P(t+1)$.
11: $t:=t+1$;
12: Update the $X^{*\prime}$;
end while
Delete all the rules number 0 in the optimal individual $X^{*\prime}$ and get a
14: sequence X^*;
15: Output the X^*;

The stop criterion is that the EA4PO will terminate after the response time cannot be improved any longer for given continuous generations or maximal generation has been achieved.

V. CASE STUDY

In this section, in order to validate the effectiveness of EA4PO, web application (WebApp) [11] is selected as the experimental case to compare our work with Xu's. Firstly Xu's work is briefly introduced. Then the WebApp case is illustrated concisely. Finally, the experimental process and result analysis are depicted in detail.

A. Xu's work

Based on software architecture described in UML SPT [10] and LQN model, performance improvement rules and the performance optimization algorithms are proposed in Xu's work. They are demonstrated in brief as follows.

① The performance improvement rules

The performance improvement rules based on LQN model proposed by Xu are divided into configuration improvement rules and design improvement rules. The configuration improvement rules are used to find the bottleneck problem caused by the unreasonable utilization or allocation of resources and eliminate this problem by means of resource reconfiguration. The configuration improvement rules contain three rules which are rules *r1*, *r2*, *r3*. In addition, design improvement rules are utilized to discover the long path problem caused by the inappropriate designs. The design improvement rules consist of rules *r4*, *r5*, *r6*, *r7*.

② The optimization algorithm

Xu designed the optimization algorithms to find the optimal solution in performance improvement space by use of the improvement rules. As shown in Fig. 2, the search process of the algorithms can be described as a search tree and divided into several rounds.

In Fig. 2, a round search is marked in the dashed rectangle. Concretely, the search starts from the root node LQN_0 extracted from the initial software architecture SA_0. Firstly, in the configuration space, the rules sequence <r1, r2, r3> are repeatedly applied to LQN_0 till the sequence <r1, r2, r3> has no improvement effect. Then, a candidate model can be got. Next, a round search for this candidate model will happen. In a round search, design improvement rules r4, r5, r6 and r7 are applied respectively to this model and a group of improved models can be obtained. Furthermore, each improved model is optimized by repeatedly applying the rule sequence < r1, r2, r3>. So, a group of candidate models can be obtained in the end of a round search. Finally, the response time of systems corresponding to these candidate models are evaluated. How to search in next round depends on the search strategy.

Based on the depths first search (DFS) strategy, Xu proposed the DFS algorithm. In DFS, the candidate model with the best response time is selected as the sole initial model in next round.

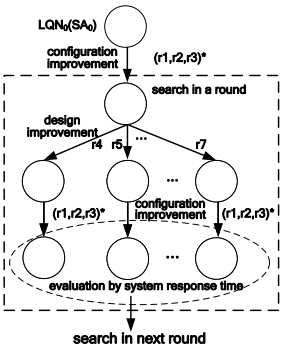

Fig. 2. The search process of Xu's algorithm

B. Case:A web application

In WebApp case, when user accesses the pages deployed on Web server (WS) by browser, WS loads these pages. Then WS will call business services deployed on application servers App1 and App2. Then, these business services will call data management services deployed on database servers DB1 and DB2. The initial LQN model of this case can be found in [11] and the response time corresponding to this model is 179.71 milliseconds.

The EA4PO and DFS algorithms are respectively applied to the WebApp case for obtain the shortest response time.

The six rules from r1 to r6 defined by Xu are used during the execution of the two algorithms.

C. Experiment

① Parameter setting

There are some parameters which are relevant to six improvement rules from r1 to r6 used in the WebApp case. To fairly compare EA4PO with DFS algorithm, the setting of these parameters in EA4PO and DFS is same. The values of these parameters shown in Table III come from Xu's paper [11].

For more convenient comparison, the optimal improvement solution can be represented as the sequence of rule numbers on the search path from root node to the node with the shortest response time of system, as far as DFS is concerned.

TABLE III. THE SETTING OF PARAMETERS RELEVANT TO SIX IMPROVEMENT RULES

No.	Name	Description	Value
1	$\Delta_{ct,\,S,\,e}$	limiting fraction for reduction in the CPU demand of entry e with the r4	0.2
2	$\Delta_{ct,\,Y,\,e,\,e'}$	limiting fraction for reduction in the calls from entry e to entry e', under the r4	0.2
3	$\Delta_{ma,\,S,\,e,\,et}$	limiting fraction in which the CPU demand of first phase of et is moved to second phase when e calls et ,under the *r5*	0.5
4	$\Delta_{ma,\,Y,\,e,\,et}$	limiting fraction in which the call number of first phase of et moved to second phase when e calls et , under the rule *r5*	0.5
5	*Mmax,WS*	The maximal multiplicity of *WS*	10
6	*Mmax,App1*	The maximal multiplicity of *App1*	5
7	*Mmax,App2*	The maximal multiplicity of *App2*	3
8	*Mmax,DB1*	The maximal multiplicity of *DB1*	10
9	*Mmax,DB2*	The maximal multiplicity of *DB2*	10
10	$U_{max,\,r}$	The predefined maximal utilization of resource	0.95
11	Δ_u	Limiting fraction for reducing resource utilization	0.9

For more strict comparison, two conditions are set. One is that each rule from r1 to r6 has the opportunity to be used. Another is that the maximal usage number of each rule for EA4PO algorithm is less than or equal to the maximal occurrence times in the improvement solution of DFS.

In addition, in EA4PO algorithm, the parameters of the weight factor, the population size, run times, evolutionary generation, crossover probability and mutation probability are set as 0.142,30, 20, 30, 0.6 and 0.3, respectively. Here, we let k be 0.01, 0.1, 0.14, 0.142, 0.2, 0.35, 0.57 respectively. And the results are better when *k* is 0.142.

② Results

The results of the performance optimization for the WebApp case by use of DFS and EA4PO algorithms are shown in Table IV and Fig. 3.

In Table IV, it is worth noting that the imprNum, totNum, imprNums and totNums rows respectively give the count of each rule usage with improvement effect, the total count of each rule usage, the count of all rules usage with improvement effect, and the total count of all rules usage. From Table IV, it can be seen that the total count of all rules usage in EA4PO (average results of 20 runs) is 15, which is less than 23 in DFS. And the count of each rule usage with improvement effect in EA4PO is equal to that of DFS except for the rule r2. From Fig. 3, the resulting response time obtained by EA4PO is 26.50 milliseconds which is better than 29.88 milliseconds obtained by DFS. In addition, the Wilcoxon rank-sum test is done to compare the resulting response time obtained by EA4PO and DFS algorithms at a 0.05 significance level. The results show that EA4PO is significantly better than DFS.

In WebApp case, it should be explained that the count of each rule usage with improvement effect in DFS is larger than that in EA4PO, and the resulting response time obtained by DFS is worse than EA4PO. It leads to such result that the order of rule usage in DFS and EA4PO algorithms is different. Our RSEF records two sequences of < r2, r2, r2, r2, r2, r3, r4, r5 > and < r2, r2, r2, r5, r2, r3, r4 >, which are obtained by deleting the rules without improvement effect from two solutions solved by DFS and the run of EA4PO nearest to the average response time 26.50 milliseconds. Therefore, EA4PO managed to achieve a better response time while using a smaller number of rules.

TABLE IV. THE USAGE INFORMATION OF RULES IN THE OPTIMAL SOLUTIONS SOLVED BY DFS AND EA4PO ALGORITHMS

		DFS	EA4PO (Average result)
r1	imprNum	0	0
	totNum	7	3
r2	imprNum	5	4
	totNum	7	5
r3	imprNum	1	1
	totNum	7	4
r4	imprNum	1	1
	totNum	1	1
r5	imprNum	1	1
	totNum	1	1
r6	imprNum	0	0
	totNum	0	1
footing	imprNums	8	7
	totNums	23	15

imprNum: the count of each rule usage with improvement effect;
TotNum: the total count of each rule usage;
imprNums: the count of all rules usage with improvement effect
totNums: the total count of all rules usage;

Two conclusions can be drawn from above results when the shortest response time was solved respectively by DFS and EA4PO algorithms. One is search range may be significantly reduced in DFS algorithm when the improvement space is divided into two sub-spaces of configuration and design, and explored by pre-defined order. The other is that the fixed order of rule usage in configuration and design space may further make the search space of DFS algorithm smaller. As a result, it is difficult for DFS algorithm to find the optimal solution. Compared with

DFS, EA4PO can explore a relatively larger space and the quality of solution can be improved.

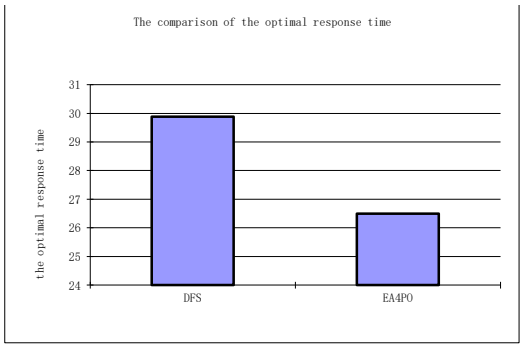

Fig. 3. The comparison of the resulting response time obtained by EA4PO and DFS algorithms

VI. CONCLUSION

In this paper, the RPOM optimization model is presented to formally describe the relation between the usage of rules and optimal solution in performance improvement space. The RPOM model helps to search the larger space by fully considering the count and order of each rule usage in the optimization process. RSEF framework is designed to support the execution of rule sequence. Furthermore, EA4PO algorithm is proposed to find the optimal performance improvement solution based on RPOM model and RSEF framework. In EA4PO algorithm, the individual encoding contributes to use performance improvement rules flexibly during the evolution. The adaptive mutation operator can learn from the heuristic information collected during the execution of rules so as to accelerate convergence of EA4PO. The fitness function considers both the extent of performance improvement and the count of rules usage with improvement effect. Experimental results show that our EA4PO algorithm can obtain better system response time and more efficient rules usage than Xu's DFS algorithm. RPOM model, RSEF framework and EA4PO algorithm proposed by this paper have certain generality and can help those rule-based software performance optimization approaches at *SA* level to search the larger space in order to improve the quality of optimization.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (No. 61305079, 61305086, 61203306, 61370078), the project of preeminent youth fund of Fujian province(JA12471), outstanding young teacher training fund of Fujian Normal University (No.fjsdjk2012083), the open fund of State Key Laboratory of Software Engineering(No. SKLSE 2014-10-02), and EPSRC Grant No. EP/J017515/1.

REFERENCES

[1].Balsamo, S., et al., "Model-based performance prediction in software development: A survey," IEEE Transactions on Software Engineering, vol.30, no.5, pp. 295-310, 2004.

[2].Koziolek, H., "Performance evaluation of component-based software systems: A survey," Performance Evaluation, vol. 67, no.8, pp. 634-658, 2010.

[3].Trivedi, K.S., "Probability & statistics with reliability, queuing and computer science applications," John Wiley & Sons, 2008.

[4].Woodside, M., et al., "Automated performance modeling of software generated by a design environment," Performance Evaluation, vol.45, no.2, pp. 107-123, 2001.

[5].Cortellessa, V., A. D Ambrogio and G. Iazeolla, "Automatic derivation of software performance models from case documents," Performance Evaluation, vol. 45, no.2, pp. 81-105, 2001.

[6].Marsan, M.A., G. Balbo and G. Conte, "Performance models of multiprocessor systems," 1986.

[7].Hermanns, H., U. Herzog and J. Katoen, "Process algebra for performance evaluation," Theoretical computer science, vol.274, no.1, pp. 43-87, 2002.

[8].Lindemann, C., et al. "Performance analysis of time-enhanced UML diagrams based on stochastic processes," Proceedings of the 3rd international workshop on Software and performance, ACM, 2002:

[9].Martens, A., et al. "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms," Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering, ACM, 2010.

[10].Tribastone, M. "Efficient optimization of software performance models via parameter-space pruning," Proceedings of the 5th ACM/SPEC international conference on Performance engineering, ACM, 2014.

[11].Xu, J., "Rule-based automatic software performance diagnosis and improvement," Performance Evaluation, vol.69, no.11, pp. 525-550, 2012.

[12].McGregor, J.D., et al., "Using arche in the classroom: One experience," DTIC Document, 2007.

[13].Cortellessa, V., et al., "A process to effectively identify "guilty" performance antipatterns," Fundamental Approaches to Software Engineering, T. Maibaum, T. Maibaum^Editors, Springer: Berlin, pp. 368-382, 2010.

[14].Trubiani, C. and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models," ACM SIGSOFT Software Engineering Notes, ACM, 2011.

[15].Cortellessa, V., A. Di Marco and C. Trubiani, "An approach for modeling and detecting software performance antipatterns based on first-order logics," Software & Systems Modeling, vol.13, no.1, pp. 391-432, 2014.

[16].Cortellessa, V. and L. Frittella, "A framework for automated generation of architectural feedback from software performance analysis," Formal Methods and Stochastic Models for Performance Evaluation, K. Wolter, K. Wolter^Editors, Springer: Berlin, pp. 171-185, 2007.

[17].Smith, C.U. and L.G. Williams. "Software performance antipatterns," Workshop on Software and Performance. 2000.

[18].Williams, L.G. and C.U. Smith, "Performance solutions: a practical guide to creating responsive, scalable software," Addison-Wesley, Reading, MA, 2001.

[19].Bondarev, E., M.R. Chaudron and E.A. de Kock, "Exploring performance trade-offs of a JPEG decoder using the DeepCompass framework," Proceedings of the 6th international workshop on Software and performance, New York, USA: ACM, 2007.

[20].Beyer, H., H. Schwefel and I Wegener, "How to analyse evolutionary algorithms,"Theoretical Computer Science, vol. 287, no.1, pp. 101-130, 2002.