Reliability-Based Design Optimization for Cloud Migration

Weiwei Qiu, Zibin Zheng, *Member, IEEE*, Xinyu Wang,
Xiaohu Yang, *Member, IEEE*, Michael R. Lyu, *Fellow, IEEE*

Abstract—The on-demand use, high scalability, and low maintainance cost nature of cloud computing have attracted more and more enterprises to migrate their legacy applications to the cloud environment. Although the cloud platform itself promises high reliability, ensuring high quality of service is still one of the major concerns, since the enterprise applications are usually complicated and consist of a large number of distributed components. Thus, improving the reliability of an application during cloud migration is a challenging and critical research problem. To address this problem, we propose a reliability-based optimization framework, named ROCloud, to improve the application reliability by fault tolerance. ROCloud includes two ranking algorithms. The first algorithm ranks components for the applications that all their components will be migrated to the cloud. The second algorithm ranks components for hybrid applications that only part of their components are migrated to the cloud. Both algorithms employ the application structure information as well as the historical reliability information for component ranking. Based on the ranking result, optimal fault-tolerant strategy will be selected automatically for the most significant components with respect to their pre-defined constraints. The experimental results show that by refactoring a small number of error-prone components and tolerating faults of the most significant components, the reliability of the application can be greatly improved.

Index Terms—Cloud Migration, Component Ranking, Fault Tolerance, Software Reliability

1 Introduction

Cloud computing enables convenient, on-demand network access to a shared pool of configurable computing resources [31]. In the cloud computing environment, the computing resources (e.g., networks, servers, storage, etc.) can be provisioned to users on-demand, like the electricity grid [5], [11]. Startup companies can deploy their newly developed Internet services to the cloud without the concern of upfront capital or operator expense [5]. However, cloud computing is not only for startups, its cost effective, high scalability and high reliability features also attracted enterprises to migrate their legacy applications to the cloud [23]. Before the migration, enterprises usually have the concern to keep or improve the application reliability in the cloud environment. Thus, reliability-based optimization when migrating legacy applications to the cloud environment is becoming an urgently required research problem.

In traditional software reliability engineering, there are four major approaches to improve system reliability: fault prevention, fault removal, fault tolerance, and fault forecasting [30]. When turning to the cloud environment, since the applications deployed in the cloud are usually complicated and consist of a large number of components, only employing fault prevention techniques and fault removal techniques are not sufficient. Another approach for building reliable systems is software fault tolerance [29], which is to employ functionally equivalent components to tolerate faults [6]. Software fault tolerance approach takes advantage of the redundant resources in the cloud environment, and makes the system more robust by masking faults instead of removing them.

Although the cloud platform is flexible and can provide resources on-demand, there is still a charge for using the cloud components (e.g., the virtual machines of Amazon Elastic Compute Cloud or Simple Storage Service). At the same time, legacy applications usually involve a large number of components, so it will be expensive to provide redundancies for each component. To reduce the cost so as to assure highly reliability in a limited budget during the migration of legacy applications to cloud, an efficient reliability-based optimization framework is needed.

In our previous work [48], [50], FTCloud is proposed to improve the reliability of newly developed cloud applications, which identifies the most significant components depending on the structure information and expert knowledge of critical components. Compared with newly developed applications, the reliability-based optimization of legacy applications has the following difficulties:

(1) The failure rate of different components in a legacy application can vary. For example, some components in the legacy application are implemented by out-dated technology and have not been well maintained. These components can have great impact on application reliability. But they may not be

- *Weiwei Qiu, Xinyu Wang, and Xiaohu Yang are with the college of Computer Science and Technology, Zhejiang University, Hangzhou, China. Email: {qiuweiwei, wangxinyu, yangxh}@zju.edu.cn;*
- *Zibin Zheng and Michael R. Lyu are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. E-mail: {zbzheng, lyu}@cse.cuhk.edu.hk*

Digital Object Indentifier 10.1109/TSC.2013.38 1939-1374/13/$31.00 © 2013 IEEE

selected as significant component by FTCloud, since FTCloud only employs structure information and does not take component failure rate information into consideration.

(2) FTCloud needs expert knowledge to manually designate critical components. However, the migration team may not be the creator of the legacy application. So it will be difficult for them to manually list the critical components. Furthermore, the number of legacy applications as well as the number of components in these applications is large; it is thus impractical to manually identify critical components.

(3) Some applications may be restricted by enterprise security polices and only part of their components can be migrated to the cloud. The component ranking and fault-tolerant strategy selection algorithms should take these hybrid applications into consideration.

For these two reasons, FTCloud is not sufficient for improving the reliability of legacy applications. We need to take advantage of all materials of the legacy applications at hand, such as application logs, source code, etc. to automatically identify the components whose failures have great impact on the application reliability. Then provide backups for them using redundant resources in the cloud to improve the application reliability.

Based on this idea, we proposed Reliability-based Optimization in Cloud environment (ROCloud), which is a component ranking framework based on historical information to identify the significant components that have great impact on application reliability, and suggest optimal fault tolerance strategies automatically. ROCloud can help the designer optimize legacy application design to get a more reliable and robust cloud application effectively and efficiently.

The contribution of this paper includes:

- This paper presents a design optimization framework for the cloud migration, named ROCloud. The main idea of this framework is first to identify significant components whose failures can have great impact on application reliability based on the application structure information and components reliability properties, and then provide fault-tolerant mechanism for these components to improve application reliability.
- ROCloud includes two ranking algorithms. The first algorithm ranks components for the applications that all their components can be migrated to the cloud. The second algorithm ranks components for hybrid applications that only part of their components can be migrated to the cloud.
- We conduct extensive experiments to evaluate the impact of significant components and their reliability properties on the reliability of the migrated application using reliability information of real-world Web services.

The rest of this paper is organized as follows. Section 2 lists the optimization challenges in cloud migration and proposes a three phase framework. Section 3 illustrates the details of the optimization framework. Section 4 shows experiments, Section 5 introduces related work, and Section 6 concludes the paper.

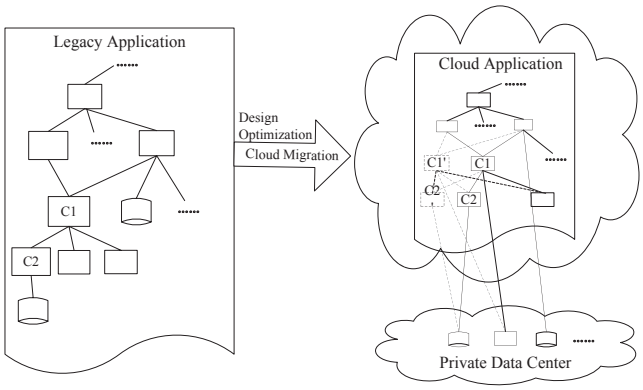


Fig. 1. Cloud Migration Example

2 Optimization Framework for Cloud Migration

2.1 Optimization Challenges in Cloud Migration

First, we use a motivating example to show the challenging problems of this paper. Enterprise A wants to reduce upfront capital investment and system infrastructure maintainance effort. The cloud computing technology satisfies these requirements. Enterprise A decides to migrate its legacy applications to an IaaS cloud, as shown in Figure 1. The legacy application consists of a number of distributed components. Ensuring reliability of the application is one of the major concerns for making the migration.

To enhance the system reliability, the designer wants to optimize the original design of legacy application by providing fault tolerance mechanisms for its components with replication techniques. When designing fault tolerance mechanisms for the components, the designer needs to consider the following problems:

(1) Some components of the legacy application may be implemented by outdated technology and suffer from high failure rates. These components can have great impact on system reliability. Replication techniques are not enough to improve the reliability. For example, providing one replication for a component with failure rate 50% can only reduce the failure rate to 25% which is still unacceptable. A better approach is refactoring, that is to adopt new technology to rewrite the component and add fault prevention logic (e.g., exception handling), which can dramatically reduce the component's failure rate. Trade-offs need to be made when considering which components should be re-factored due to cost constraints.

(2) The legacy application may consist of a large number of components. It is too expensive to deploy

alternative replicas for all the components, since there are costs for using cloud resources (e.g., the virtual machines). To make trade-offs between costs and reliability, the designer chooses to tolerate faults of the most important components, whose failures have great impact on the whole system. However, it is not easy to identify which components have greater impact on system reliability, because:

- The reliability properties of each component may be very different. Some components may already have fault prevention logic (e.g., error checking, exception handling, etc.) and thus are more reliable than others.
- Failures of different components can have different impacts on the system. Components fulfilling critical tasks (e.g., payment) are taken as critical components, while other components accomplishing non-critical tasks (e.g., providing decorative pictures on web pages) are taken as non-critical ones [48]. Failures of critical components have greater impact on the system than failures of non-critical components.

These two characteristics should be considered in combination. A failure-prone non-critical component may have little impact on overall system reliability, while a component for critical task may be carefully designed and already have low enough failure rate. The straightforward approach to only consider components with high failure rates or fulfilling critical tasks as important components may not lead to an optimal solution.

(3) Some applications are restricted by enterprise security polices and only part of their components can be migrated to the cloud. For these hybrid applications, the components which are kept in the private data center are potentially important components and they can only use resources in the private data center for fault tolerance.

(4) There are a number of fault tolerance strategies. The cloud platform itself may also provide recovery approaches such as virtual machine restart. Different strategies have different overheads and costs. It is a challenging task for the designer to find out the optimal fault tolerance strategies for the significant cloud components.

To address the above problems, we first analyze the legacy application to collect the reliability properties and application structure information. Then, we proposes two significant component ranking algorithms in Section 3.2. At last an optimal fault tolerance strategy selection algorithm is presented in Section 3.3.2, which suggests optimal fault tolerance strategies for components with different constraints.

2.2 Optimization Framework

Figure 2 shows the overview of our reliability optimization framework (named ROCloud), which includes three

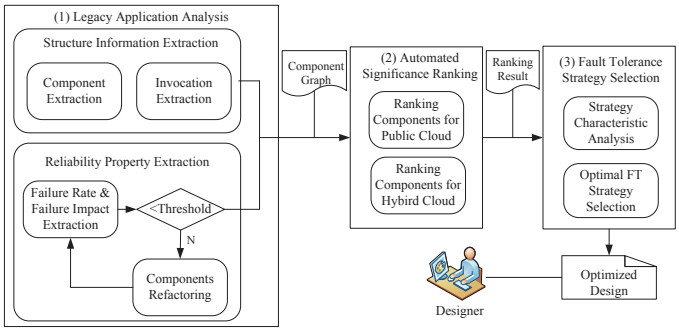


Fig. 2. Overview of the Optimization Framework

phase: (1) legacy application analysis automated significance ranking and (2) fault tolerance strategy selection. The processes of each phase are as follows:

(1) Both structure and failure information are extracted from during the legacy application analysis phase. The structure information extraction consists of two sub-processes: component extraction and invocation extraction. The failure information including failure rate and failure impact are collected from the execution logs and test results of the legacy application. Components with a failure rate higher than the threshold will be re-factored, and their reliability property will be updated.A component graph is built for the legacy application based on the structure as well as the failure information.

(2) In the automated significance ranking phase, two algorithms are proposed for ordinary applications that can be migrated to public cloud and hybrid applications that need to be migrated to hybrid cloud, respectively.

(3) The performance, overhead, and cost of various fault tolerance strategy candidates are analyzed and the most suitable fault tolerance strategy is selected for each significant component based on its predefined constraint.

Section 3 will introduce the technical details of the legacy application analysis, component ranking algorithms and the optimal fault tolerance strategy selection algorithm.

3 Approach

The ROCloud aims to quantify the importance of each component from the application reliability aspect based on available information such as application structure, component invocation relationships, components' reliability properties, and so on. Thus, the legacy applications need to be analyzed to collect the information for ranking.

3.1 Legacy Application Analysis

3.1.1 Structure Information Extraction

The structure information includes components and the invocation information. The components are extracted

from legacy applications by source code and documentation analysis. The invocation information such as invocation links and invocation frequencies can be identified from application trace logs. Source codes and documentations are useful supplementary materials besides trace logs. All the information are represented in a component graph.

The component graph is modeled as a weighted directed graph G, which combines together the information of application structure and the invocation relationships among components [48], [50]. Node c_i in graph G represents a component and each component has a nonnegative significance value $V(c_i)$, failure rate $f(c_i)$ and failure impact $p(c_i)$. A directed edge e_{ij} from node c_i to node c_j represents component c_i invokes c_j, the total number of which is denoted by q_{ij}. Each edge e_{ij} in the graph has a nonnegative weight value w_{ij}, which can be calculated by:

$$w_{ij} = \frac{q_{ij}}{\sum_{j=1}^{n} q_{ij}}, \tag{1}$$

where n is the number of components. The range of weight value is [0,1]. If there is no edge from c_i to c_j, which means that c_i does not invoke c_j, $q_{ij} = 0$ and thus $w_{ij} = 0$. In this way, w_{ij} has a larger value if component c_j is invoked more frequently by component c_i compared with other components invoked by c_i.

For a component graph which contains n components, an $n \times n$ transition probability matrix W can be obtained by employing Eq. (1). Each entry in the matrix is the value of w_{ij}. In the case that a node c_i has no outgoing edge, $w_{ij} = \frac{1}{n}$. For $\forall i$, the transition probability matrix W satisfies:

$$\sum_{j=1}^{n} w_{ij} = 1. \tag{2}$$

3.1.2 Reliability Property Extraction

(1) Component Failure rate and Failure Impact Collection

The failure rate and failure impact information can be collected from the execution logs or test results of legacy applications. Failure rate $f(c_i)$ of component c_i can be calculated by:

$$f(c_i) = \frac{\mu(c_i)}{\sum_{j=1}^{n} q_{ji}}, \tag{3}$$

where $\mu(c_i)$ is the total times that component c_i failed, and the sum represents the total times that c_i has been invoked. As mentioned in Section 1 and Section 2.1, the failures of different components can have different impact on system failure. In [48], prior knowledge provided by the system designer was employed, and the components were divided into two sets: critical and non-critical. If a critical component fails, the application will fail too, while the failure of a non-critical component will not cause an application failure. However, this method depends greatly on expert knowledge of the designer, and requires that the designer has sophisticated understanding of the application business logic. However, getting acquainted with business logic of the legacy application can increase the the cost of the migrating process. At the same time the dichotomy of component failure impact is not accurate enough. Thus, we use a value calculated from statistics to estimate the failure impact $p(c_i)$ of component c_i:

$$p(c_i) = \begin{cases} \mu(a|c_i)/\mu(c_i) & \text{if } \mu(c_i) \neq 0, \\ 0 & \text{if } \mu(c_i) = 0. \end{cases} \tag{4}$$

where $\mu(a|c_i)$ is the failure times of the legacy application when failure of component c_i occurs.

(2) Component Refactoring

During the migration, we found that some legacy components suffer from high failure rates. These almost "dead" components are the bottle-necks of the application reliability and adopting only fault-tolerant strategies based on replication is not enough to improve their reliability. Thus component refactory which is also known as re-engineering or re-implementation is needed. Almonaies et al. listed the characteristics with which the legacy systems are especially applicable to refactoring from the view point of service-oriented re-engineering [2].

In this paper, the main optimization goal is reliability, so a more straightforward way is employed to determine which components should be re-implemented: components with failure rates greater than a threshold. The selection of the threshold is dependent on project budget and the target application failure rate, since extra development and testing effort is required for component re-implementation. The impact of threshold is tested in experiments in Section 4.5. After refactoring, the component failure rates will be estimated based on test results, and the component reliability property data set will be updated.

3.2 Automated Significance Ranking

Based on the component graph, two component ranking algorithms are proposed in this section. The first algorithm ranks components for ordinary applications where all their components can be migrated to the cloud. The second algorithm rank components for hybrid applications which can be partly moved to the cloud.

3.2.1 ROCloud1: Component Ranking for Ordinary Applications

In a distributed application, the failures of the components which are frequently invoked by many other components tend to have greater impact on the system compared with the components which are rarely invoked by others. Thus these components are considered to be more important from the reliability aspect and should be ranked at the front of component list. Inspired by the PageRank algorithm [9], we propose an algorithm to calculate the significance value of each component

of the migratory application employing the component invocation relationships and reliability properties. Based on the component graph and component reliability information, the component ranking algorithm includes the following steps:

1. Initialize by randomly assigning a numerical value between 0 and 1 to each component in the component graph.

2. Compute the significance value for a component c_i by:

$$V(c_i) = \frac{1-d}{n} f(c_i)p(c_i) + d \sum_{k \in N(c_i)} V(c_k) w_{ki}, \quad (5)$$

where n is the number of components in the application, and $N(c_i)$ is the set of components which invoke component c_i. The parameter d ($0 \leq d \leq 1$) in Eq. (5), also known as damping factor, is employed to adjust the significance values derived from other components, and is generally set around 0.85. $f(c_i)$ is the component failure rate which is obtained by employing Eq. (3). $p(c_i)$ is the component failure impact on the application, which is calculated by Eq. (4). Consequently, the significance value of c_i is composed of the basic value of itself (i.e., $\frac{1-d}{n} f(c_i)p(c_i)$) and the derived values from the components that invoked c_i. By Eq. (5), a component c_i will have a larger significance value if the values of $f(c_i)$, $p(c_i)$, $|N(c_i)|$, $V(c_k)$, and w_{ki} are larger, indicating that component c_i is invoked by a lot of other significant components frequently and tends to cause application failures.

In vector form, Eq (5) can be written as:

$$\begin{bmatrix} V(c_1) \\ \vdots \\ V(c_n) \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + dW^t \begin{bmatrix} V(c_1) \\ \vdots \\ V(c_n) \end{bmatrix}, \quad (6)$$

where

$$x_i = \frac{1-d}{n} f(c_i)p(c_i). \quad (7)$$

3. The significance values can be calculated either iteratively or algebraically. The iterative method is repeating the computation until all significance values become stable, that is, $|V(t+1) - V(t)| < \epsilon$, for some small ϵ.

Since weight matrix W is a stochastic matrix which is shown in Section 3.1.1, Eq. (6) can also be solved by computing the eigenvector with eigenvalue equal to 1.

With the above approach, the significance values of the components can be calculated by considering the application structure information, the invocation relationships, and the knowledge of component reliability properties in combination. A component with a larger significant value is considered to be more significant. The failures of these significant components will have great impact on other components and thus tend to cause application failures.

3.2.2 ROCloud2: Component Ranking for Hybrid Applications

Hybrid applications are the applications that only part of their components can be migrated to the public cloud. Restricted by enterprise security policies, some components (e.g., the components related to custom private data, components implementing core business logic, etc.) of these applications cannot be opposed to third-parties and need to be kept in the private data center. The decision that which components should be kept local mainly depends on the enterprise security policies and the contracts with customs. For these hybrid applications, the components which are kept in the private data center are potentially important components. Inspired by the work of [44], relative importance are calculated for each component. The components of a hybrid application are divided into two sets by their nature. One set for the components deployed in a private data center, denoted as P, and the other for the components moved to the cloud, denoted as C.

For each component c_i, the significance value can be calculated by:

$$V(c_i) = (1-d)\rho + d \sum_{k \in N(c_i)} V(c_k) w_{ki}, \quad (8)$$

where

$$\rho = \begin{cases} \frac{1}{|P|} & \text{if } c_i \in P, \\ \frac{f(c_i)p(c_i)}{|C|} & \text{if } c_i \in C. \end{cases} \quad (9)$$

$|P|$ and $|C|$ are the numbers of components in the private data center and the cloud respectively, $|P| + |C| = n$. $f(c_i)$ is the component failure rate which is obtained by employing Eq. (3). $p(c_i)$ is the component failure impact on the application, which is calculated by Eq. (4). When $|C| = n$, RCloud2 degrades to ROCloud1. By employing Eq. (8) and following the steps listed in Section 3.2.1, the significance values of all the components can be calculated.

Based on the ranked list of components with significance values, the top k ($1 \leq k \leq n$) components can be identified by the designer at the design optimization stage of migration. Various techniques (e.g., fault tolerant techniques or recovery approaches which is illustrated in Section 3.3) can be employed to improve the reliability of these components, and therefore improving the application reliability. The greater the value of top k is, the more reliable the application tends to be, while at the same time the more cloud resources are needed. The impact of top k on application reliability is shown in detail in Section 4.7.

3.3 Fault Tolerance Strategy Selection

3.3.1 Strategy Characteristic Analysis

Software fault tolerance is widely adopted for critical systems (e.g., airplane flight control systems, nuclear power station management systems, etc.). At the same time, a cloud platform also provides approaches such

as virtual machine restart, virtual machine migration, etc. to improve components reliability. By employing these techniques to provide functionally equivalent components, the component failures can be tolerated and thus the overall system reliability can be increased. Three well-known software fault tolerance strategies as well as the approaches taking advantage of cloud platform features are introduced in the following with formulas for calculating the failure rate, response time and resource cost.

(1) Software fault tolerance strategies

Recovery Block (RB) [36], **N-Version Programming (NVP)** [6] and **Parallel** are three widely used strategies in software fault tolerance. Their features and failure rate calculation have been summarized in [48], [50]. In this paper, the formulas to calculate response time t and resource cost r are given in Eq.(8)-(10) in Table 1 (the equations to calculate failure rate f are also listed for reference). In these equations, n is the number of redundant components, f_i is the failure rate, t_i is the response time, and r_i is the resource allocated for the i^{th} component.

Since RB strategy invokes standby component sequentially when the primary component fails, its response time is the summation of the execution time of all failed versions and the first successful one. NVP strategy needs to wait for all n responses from the parallel invocations to determine the final result, thus its response time depends on the slowest version. While Parallel strategy employs the first returned response as the final result, its response time is the minimum one of all replications. So it can be concluded from Eq. (8)-(10) that the response time performance of RB is generally worse than that of NVP, which in turn is worse than that of the parallel strategy. Since NVP and Parallel use parallel component invocations and all the resources need to be allocated before the execution, while in RB extra resources will be allocated only when the primary component fails, the required resources of NVP and Parallel are much higher than those of RB. All three strategies can tolerate crash faults, and NVP strategy can also mask value faults [48], [50].

(2) Strategies based on cloud features

Cloud platforms often provide approaches such as virtual machine restart, virtual machine migration [1], [25], etc. to improve reliability. These approaches can also tolerate crash faults. The strategy based on virtual machine (VM) restart is similar to the RB strategy, but it introduces overhead (OH in Eq.(11)) to initialize virtual machines and thus its response time is larger than that of RB, as shown in Eq.(11) in Table 1. However, VM restart requires no extra resources, since it is based on restart. The strategy based on virtual machine migration can tolerate non-transient hardware crash faults such as disk failures but it can also increases the response time. Strategy based on node restart must be used carefully, since the restart may affect other components.

TABLE 1
Formulas for Fault Tolerance Strategies

Fault Tolerance Strategy	Formulas	
Recovery Block	$f = \prod_{i=1}^{n} f_i$ $t = \sum_{i=1}^{n} t_i \prod_{k=1}^{i-1} f_i$ $r = \sum_{i=1}^{n} r_i \prod_{k=1}^{i-1} f_i$	(8)
NVP	$f = \sum_{i=(n+1)/2}^{n} F(i)$ $t = \max t_i$ $r = \sum_{i=1}^{n} r_i$	(9)
Parallel	$f = \prod_{i=1}^{n} f_i$ $t = \min t_i$ $r = \sum_{i=1}^{n} r_i$	(10)
VM Restart	$f = \prod_{i=1}^{n} f_i$ $t = \sum_{i=1}^{n} (OH + t_i) \prod_{k=1}^{i-1} f_i$ $r = max r_i$	(11)

In summary, strategies based on cloud features can also improve components reliability by tolerating crash faults. They have much lower demand on extra resource compared to the software fault tolerance strategies, while they have considerable overheads which can increase the response time.

Different strategies have different resource requirement and different effects on response time. RB strategy can affect the response time and resource allocation if there is a failure. While the parallel and NVP strategies have little effect on the response time but will affect the resource allocation in all cases. The virtual machine restart strategy will not affect resource allocation but can affect the response time if there is a failure. Employing a suitable fault tolerance strategy for the significant components can help achieve optimal resource allocation while improving application reliability. Each fault tolerance strategy has a number of variations, thus selecting an optimal strategy for each significant component is time consuming. An automatic optimal fault tolerance strategy selection algorithm is therefore required to reduce the workload of application designers.

In summary, four candidates are employed for fault tolerance in this paper, which includes recovery block, N-version programming, parallel, and virtual machine restart. These strategies can be employed to tolerate crash and value faults. Other types of fault tolerance mechanisms can be added to ROCloud without fundamental changes.

3.3.2 Automatic FT Strategy Selection

The software fault tolerance strategies have a number of variations based on different redundant component configurations [48], [50]. Analogy can be made to strategies based on cloud features (e.g., restart times). These variations are candidates for each significant component in the application, and the optimal FT strategy selection algorithm introduced in [48], [50] can be employed to identify the optimal one.

First, the aggregated failure rate f, response-time t, and the resource cost r of each fault tolerance strat-

egy candidate are calculated by employing Eq.(8)-(11) in Table 1. And the strategies which could not satisfy the response-time constraints will be removed. Second, list the Top-K significant components according to the descending order of their significance value. Third, the strategy with minimum resource cost will be selected for each of the components as their initialization strategy to make sure all of them are fault-tolerant. Then for each component, select the candidate with the lowest aggregated failure rate the optimal one. By repeating the last step until it meets the user resource cost constraints, the reliability-based design optimization can be achieved.

Currently, each component in the cloud is considered as independent and the fault tolerant strategy selection is carried out separately on component basis. However, in practice, some components can be interrelated, and coordinating among these components has the potential of providing fault tolerance with lower cost. But the coordinating process requires information of failure dependency and impact among the interrelated components, which is beyond the scope of this paper. In the future work, we will study failure dependency on interrelated components and provide more sophisticated fault-tolerant selection model.

4 EXPERIMENTS

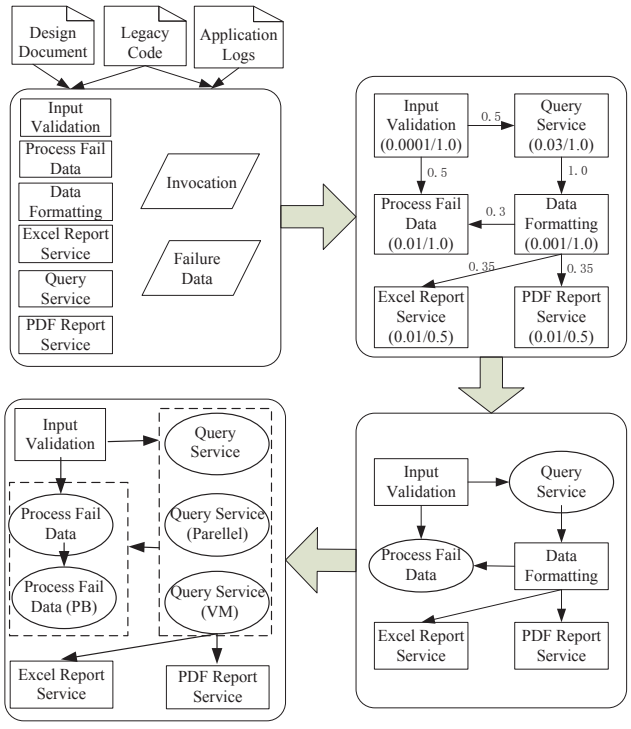


Fig. 3. Case Study

4.1 Case Study

A simple case of migrating a reporting application to the cloud is given in this section to illustrate the process of ROCloud, as shown in Figure 3.

- Step 1 Legacy application analysis: First, browse source code and documents to identify the modules/components of the application, which include Input Validation, Process Fail Data, Data Formatting, Query Service, Excel report Service and PDF report service. Then extract the invocation and reliability information from the application logs. Based on these information, a component graph is built. The number pairs in each component denote the failure rate and failure impact, and the weight on each line is calculated by Eq. (1).
- Step 2 Significance Ranking: The significance value of each component is calculated by employing the algorithms introduced in Section 3.2 on the component graph built in Step1. Components are ranked according to their significance values, and the most significant components are selected (Query Service and Process Fail Data in this case).
- Step 3 Fault tolerance strategy selection: Optimal fault tolerant strategies are provided for the Query Service and Process Fail Data components, with respect to their time and cost constraints. And the optimized design is shown in the left bottom of Figure 3, each service instance will be deployed in separate virtue machines.

4.2 Experimental Setup

The significant component ranking algorithm is implemented by C++ language. To study the performance of reliability improvement, five approaches are compared, which are:

- **NoFT**: No fault tolerance strategy is employed.
- **RandomFT**: Fault tolerance strategies are employed to mask faults of randomly selected K percent components.
- **FTCloud**: FTCloud is a component ranking framework for cloud applications presented in our previous work [48], [50] which ranks the components by only employing the structure information of the application. Fault tolerance strategies are employed to mask faults of top K percent components of the ranked list.
- **ROCloud1**: Fault tolerance strategies are employed to mask faults of the top K percent significant components. The components are ranked by employing the component ranking algorithm, which considers the structure information as well as the prior knowledge of component reliability properties such as failure rate and failure impact.
- **ROCloud2**: The components related to custom private data are kept locally in ROCloud2, which is one of the best practice used by enterprises. Fault tolerance strategies are employed to mask faults of the top K percent significant components relative to the local components. The components are ranked by employing the component ranking algorithm for hybrid applications.

- **AllFT**: Fault tolerance strategies are employed for all components.

The internal structures of software programs (e.g., call graphs for procedural code, class collaboration graphs, etc.) are shown to exhibit approximate scale-free properties in several previous work [12], [18]. Thus we use Pajek [7] to generate scale-free directed component graphs for experimental studies.

Each component in a cloud-based application can be considered as a separate Service. Thus we use the failure rate data of real-world services to simulate that of components. We deployed 150 different services on PlanetLab, and recorded the invocation times and failures of each service which was invoked and tested from 100 different locations around the world. The failure rates were calculated based on the log, and assigned randomly to the nodes of component graphs generated by Pajek [7]. When a component was selected to be fault-tolerant, the fault tolerance strategy determination algorithm was employed to automatically select the optimal fault tolerance strategy for tolerating faults. If a fault tolerance strategy was applied for this component, the component failed only when the primary and back up copies all failed. Random walk on the component graph is used to simulate the execution of each application. 10000 sequences are generated by random walk for each experimental setting to guarantee each component will be covered. The sequence execution is considered as failed if an invoked component fails and no fault tolerant strategy was provided or the whole strategy failed. All the sequences are executed for 200 times and the application failure rate is collected based on the execution result.

4.3 Performance Comparison

With the above settings, six types of fault tolerance mechanisms (i.e. NoFT, RandomFT, FTCloud, ROCloud1, ROCloud2, and AllFT) were applied on these invocation sequences, each for 200 times, and the average results are reported in Table 2.

In Table 2, Top-K (K = 10%, 20%, and 30%) indicates that fault tolerance mechanisms are applied for K percent components (K percent most significant components in FTCloud, ROCloud1 and ROCloud2; K percent randomly selected components in RandomFT). The numbers of components in applications (represented by Node Numbers) increase from 100 to 10000. The experimental results in Table 2 show that:

- Among the four approaches, AllFT provides smallest failure rate, which means the application is the most reliable with all its components being fault-tolerant. NoFT performs the worst, which provides the highest failure rate, because no fault tolerance strategy is provided for the components.
- Compared with RandomFT, FTCloud obtains better failure rate performance in all experimental settings. FTCloud identifies significant components based on the structure information. Components which are invoked most frequently are considered to be significant, and their failures have greater impact on the application. The experimental result indicates that tolerating failures of these components can achieve better system reliability than tolerating failures of randomly selected components.
- Compared with FTCoud, both ROCloud1 and ROCloud2 obtain better average failure rate in all experimental settings. Besides structure information, ROCloud also considers component reliability in component ranking. The experimental result shows that the component ranking algorithms of ROCloud achieves more accurate results when taking advantage of the prior knowledge of component reliability as well as the system structure information in combination.
- Since the components kept in the private data center are usually critical components in essence, ROCloud2 actually take advantage of simple business logic information in addition than ROCLoud1, which brings about better accuracy in component ranking and thus achieves lower failure rate.
- With the increase of $threshold$ from 1% to 10% the average failure rate of all the approaches are increased. Because only components with failure rate higher than $threshold$ are re-factored, a larger $threshold$ indicates a higher average component failure rate, which leads to a higher application failure rate.
- With the increase of the node number from 100 to 10,000, the average failure rate of all approaches increases (even ALLFT), since large-scale systems are prone to frequent failures, which is also presented by [33]. ROCloud1 and ROCloud2 can consistently provide better performance compared with RandomFT and FTCloud with different node numbers, indicating that ROCloud1 and ROCloud2 can identify the components which have great impact on application reliability, and by tolerating faults of the these components, the application reliability can be greatly improved for different sizes of cloud applications.

4.4 Comparison of FT Strategies

We introduced four types of fault tolerant strategies in Section 3.3.1. The aggregated failure rate, response time, and the resource cost of each fault tolerance strategy are shown in Table 1. Among the four strategies, NVP has the most resource cost, so it is mainly used to tolerate value faults, while the other strategies are often used to tolerate crash faults. However, the VM restart strategy has have considerable time overhead, which can not satisfy the time constraint of components in this experiment. Since it requires no extra resource, the VM restart strategy is used as an supplementary strategy to Parallel and Recovery Block (RB) in this experiment. To compare

TABLE 2
Performance Comparison of Average Application Failure Rate

Node Number	Methods	Threshold = 0.01			Threshold = 0.04			Threshold = 0.1		
		Top10%	Top20%	Top30%	Top10%	Top20%	Top30%	Top10%	Top20%	Top30%
100	NoFT	0.00261	0.00254	0.00252	0.00258	0.00253	0.00255	0.00602	0.00609	0.00601
	RandomFT	0.00251	0.00243	0.00240	0.00252	0.00245	0.00241	0.00600	0.00597	0.00586
	FTCloud	0.00167	0.00190	0.00229	0.00252	0.00236	0.00122	0.00598	0.00577	0.00262
	ROCloud1	0.00167	0.00040	0.00030	0.00248	0.00037	0.00033	0.00213	0.00205	0.00084
	ROCloud2	0.00108	0.00016	0.00002	0.00165	0.00016	0.00008	0.00194	0.00188	0.00017
	AllFT	0.00001	0.00001	0.00000	0.00000	0.00003	0.00002	0.00002	0.00002	0.00003
1000	NoFT	0.01054	0.01055	0.01038	0.01350	0.01349	0.01346	0.01808	0.01816	0.01784
	RandomFT	0.00946	0.00737	0.00710	0.01160	0.00931	0.00869	0.01508	0.01451	0.01279
	FTCloud	0.00835	0.00528	0.00409	0.01068	0.00661	0.00464	0.01425	0.00967	0.00788
	ROCloud1	0.00538	0.00208	0.00109	0.00593	0.00286	0.00137	0.00840	0.00404	0.00197
	ROCloud2	0.00462	0.00176	0.00073	0.00495	0.00239	0.00101	0.00703	0.00373	0.00162
	AllFT	0.00005	0.00005	0.00005	0.00007	0.00008	0.00009	0.00015	0.00019	0.00016
10000	NoFT	0.01186	0.01194	0.01191	0.01418	0.01431	0.01425	0.02216	0.02227	0.02214
	RandomFT	0.01069	0.00971	0.00876	0.01309	0.01173	0.01071	0.02036	0.01899	0.01612
	FTCloud	0.01024	0.00759	0.00549	0.01211	0.00876	0.00633	0.01924	0.01371	0.00934
	ROCloud1	0.00741	0.00395	0.00280	0.00832	0.00462	0.00300	0.01063	0.00567	0.00364
	ROCloud2	0.00703	0.00380	0.00263	0.00800	0.00435	0.00276	0.01006	0.00537	0.00336
	AllFT	0.00013	0.00013	0.00013	0.00015	0.00016	0.00015	0.00057	0.00055	0.00050

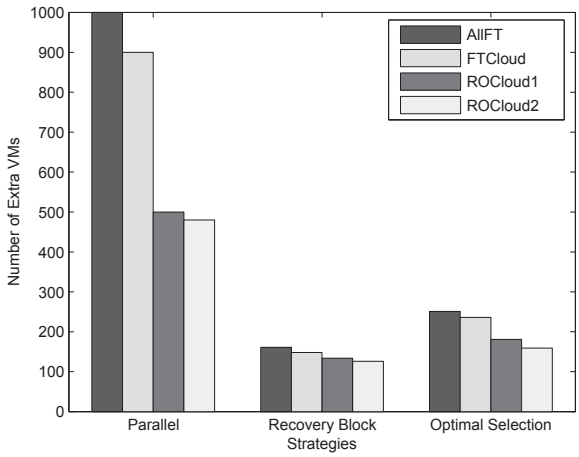
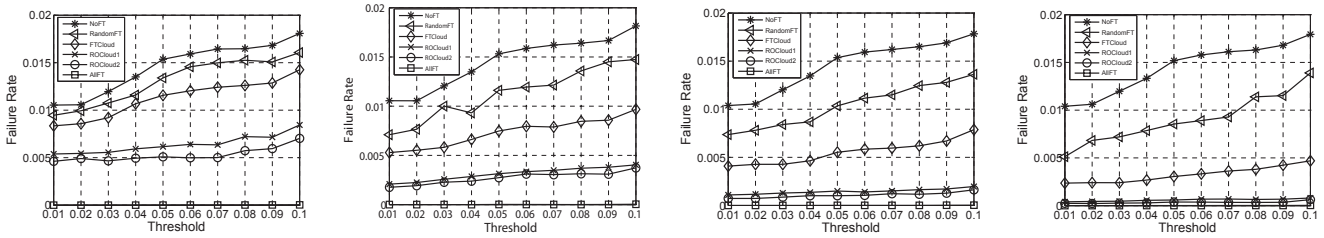


Fig. 4. Required Resources of Different FT Strategies

the performance of Parallel, RB and the optimal selection between Parallel and RB, we collected the response time and failure rate of 150 services deployed on PlanetLab. Following the experimental setup instructions presented in Section 4.2, we assign the response time and failure rate pair randomly to a 1000-node component graph. The average response time of all components is used as the time constraint in optimal fault tolerant strategy selection. The number of redundant components in each strategy is set as 1.

Figure 4 shows the required resources of different fault tolerant strategies for AllFT, FTCloud and ROCloud1 and ROCloud2 to achieve the same application reliability. As shown in Figure 4:

- When parallel strategy is adopted, AllFT needs 1000 extra VMs since it provides backup copy for each component, while FTCloud requires 900 extra VMs to achieve the same reliability level, ROCloud1 needs 500, and ROCloud2 needs 480. The result is consistent with that shown Figure 7. When the top 90% components are selected by FTCloud whereas the top 50% selected by ROCloud1 and top 48% selected by ROCloud2 are provided with fault-tolerant strategy, they achieve the same reliability level as AllFT.
- The Recover Block (RB) strategy reduces the requirement of extra resources dramatically. Since extra resources are consumed when the primary components fails. However, the response time of the failed components will increase. In this experiment, the total increased response time of failed components is 742s for AllFT, 736s for FTCloud and 720s for ROCloud1 and 712s for ROCloud2.
- Taken the average response time as time constraint, components with lower response time ($2 \times responsetime <= averageresponsetime$) take RB as their fault-tolerant strategy. Components with longer response time ($2 \times responsetime <= averageresponsetime$)take parallel strategy. The resource requirement falls between the parallel and RB strategy. The total increased response time is 177s for both AllFT and FTCloud, 174s for ROCloud1 and 171s for ROCloud2.
- The selection of different strategies is a trade-off between cost and application service quality (e,g. response time). The parallel strategy provides the shortest average response time but costs the most, while the RB strategy has the longest average response time but cost the least. Optimal selection falls between the two. The cost and response time benefit of optimal selection can be affected by the constraint factor (time constraint in this experiment). A higher constraint (shorter response time in this case) leads



(a) Top-K = 10% (b) Top-K = 20% (c) Top-K = 30% (d) Top-K = 40%

Fig. 5. Impact of Threshold

to higher resource cost, and vice versa.

- For all the strategy selection settings, ROCloud1 and ROCloud2 require fewer resources than AllFT and FTCloud to achieve the same application reliability level. Furthermore, ROCloud provides better application service quality with the same resource consumption while maintaining the same application reliability level. As shown in the figure, ROCloud1 with optimal selection costs almost the same amount of resources as AllFT with PB strategy, and ROCloud2 with optimal selection costs almost the same as FTCloud with PB strategy. While with optimal selection, both ROCloud1 and ROCloud2 have much shorter increased response time (174s and 171s, respectively) than AllFT (742s) and FTCloud (736s).

4.5 Impact of Threshold

In the component ranking approach ROCloud1 and ROCloud2, the parameter $threshold$ determines which components should be re-factored. The value of $threshold$ is set as 0.01 to 0.1 with a step value of 0.01 in this experiment. Four groups of experiments are conducted with Top-K values (k=10%, 20%, 30% and 40%) and the node number is set as 1000. Figure 5 shows the experimental results:

- With the increase of $threshold$ value from 0.01 to 0.1, the application average failure rate also increases, indicating that a higher average failure rate of components can lead to a higher application failure rate. In all the four figures with different Top-K value settings, ROCloud1 and ROCloud2 perform better than RandomFT and FTCloud.
- With the increase of Top-K value from 10% to 40%, the application failure rate becomes smaller, since faults of more significant system components are masked by the fault tolerance mechanism. This indicates that a larger K can relieve the increased application failure rate caused by a larger $threshold$.

The value of $threshold$ not only affects the application failure rate, but also affects the migration cost. A smaller $threshold$ means more components need to be re-factored. Table 3 shows the detailed results when $threshold$ increases from 0.01 to 0.1, the number of components that need to be re-factored. The refactoring of a component requires extra effort on components logic extraction, re-design, implementation and testing, which can increase the cost of migration. Thus the determination of value $threshold$ is a trade-off between cost and application reliability.

TABLE 3
Number of Refactored Nodes for Different Threasholds

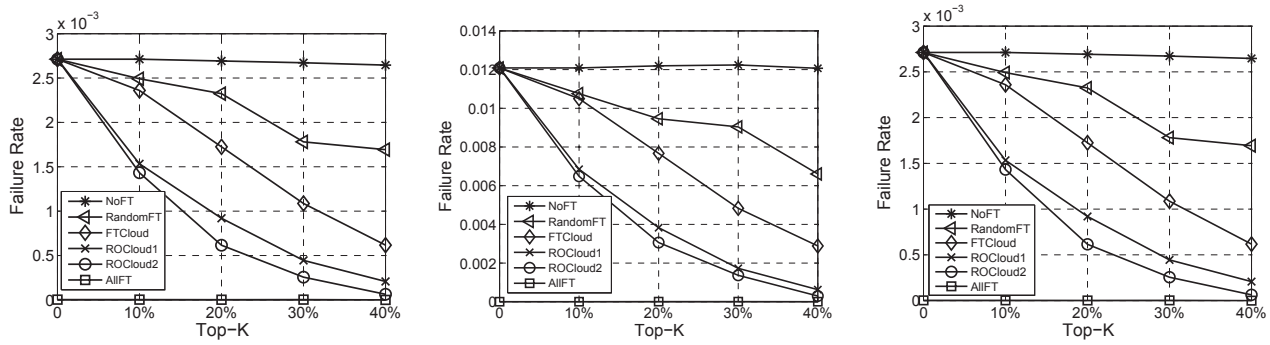
Threasholds	Node Numbers				
	100	500	1000	5000	10000
0.01	12	49	95	595	1292
0.02	8	39	69	417	957
0.03	6	27	67	365	837
0.04	4	25	53	330	758
0.05	3	19	47	287	667
0.06	3	19	40	255	574
0.07	3	19	39	225	522
0.08	3	17	36	211	486
0.09	3	17	34	199	466
0.1	3	17	31	188	451

4.6 Impact of Component Failure Impacts

Component Failure Impact (CFI) is defined by Eq. (4) in Section 3.1.2, which indicates the probability that the component failure may cause the application failure. For example, CFI = 1 means any component failure can cause the application failure. To study the impact of CFI on the application reliability, we compare NoFT, RandomFT, FTCloud, ROCloud1, ROCloud2 and AllFT under average CFI settings of 0.1, 0.5 and 1, respectively. The node number in this experiment is 1000. The $threshold$ is set as 0.04, which is the average component failure rate before refactoring in the experiment. Figure 6 shows the experimental results of cloud application failure rate under different Top-K settings.

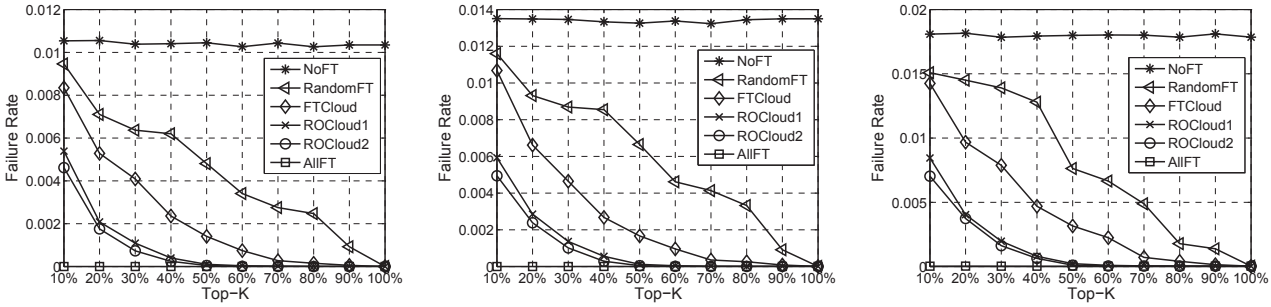
Figure 6 illustrates that:

- As shown in Figure 6(a) to Figure 6(c), ROCloud1 and ROCloud2 outperform FTCloud and RandomFT in Top-K settings from 10% to 40%. With the increase of Top-K value, the application failure rate of ROCloud1 and ROCloud2 decrease faster than those of FTCloud and RandomFT, indicating that ROCloud1 and ROCloud2 have more effective use of the redundant components than FTCloud and RandomFT.



(a) Average CFI = 0.1 (b) Average CFI = 0.5 (c) Average CFI = 1

Fig. 6. Impact of Component Failure Impacts



(a) Threshold = 0.01 (b) Threshold = 0.04 (c) Threshold-K = 0.1

Fig. 7. **Impact of Top-K**

- When the components have a greater average CFI, the application failure rate is also larger, since the components with larger CFIs mean their failures have greater impact on the application. This is consistent with the definition of CFI in Eq. (4). On the other hand, as shown in Figure 6(a) to Figure 6(c), the curves have similar tendencies, which indicates that under different CFI settings, ROCloud1 and ROCloud2 achieve better performance compared with FTCloud and RandomFT.

4.7 Impact of Top-K

To study the impact of the parameter Top-K on the application reliability, we compare NoFT, RandomFT, FTCloud, ROCloud1, ROCloud2 and AllFT according to different Top-K value settings. The node number in this experiment is 1000. Figure 7 shows the experimental results of application failure rate under three different $threshold$ settings: 0.01, 0.04 and 0.1.

Figure 7 shows that:

- Under different $threshold$ settings (i.e., 0.01, 0.04, and 0.1), ROCloud1 and ROCloud2 outperform FTCloud and RandomFT from Top-K = 10% to Top-K = 90% consistently. When Top-K = 100%, since fault tolerance strategies are applied to all the components, the performance of the three approaches are the same.
- With the increase of Top-K value, the failure rate of ROCloud1 and ROCloud2 decreases much faster than those of FTCloud and RandomFT, indicating that ROCloud1 and ROCloud2 provide more accurate ranking results of significant components. By tolerating faults of the components suggested by ROCloud, the application reliability can be improved greatly.
- ROCloud1 and ROCloud2 obtain a smaller application failure rate than FTCloud and RandomFT approaches consistently under different Top-K value settings. As shown in Figure 7(a) to Figure 7(c), when Top-K=50% for ROCloud1 and Top-K=48% forROCloud2 can almost get the same application failure rate as AllFT (which has the same effect as Top-K=100%). FTCloud can achieve roughly the same performance until Top-K = 90%. The results indicate that ROCloud1 and ROCloud2 can improve the application reliability greatly by tolerating the important part of the application components.
- With the increase of $threshold$ from 0.01 to 0.1, the application failure rate of all three approaches become larger. This is because the average failure rate of components increases since a larger $threshold$ indicates fewer components will be refactored. A larger Top-K value is required to achieve good application failure rate performance under large $threshold$ settings. The experimental results in

this section and Section 4.5 show that Top-K and $threshold$ are complementary parameters. Designers can choose the one which costs less to improve the application reliability.

The above experimental results show, again, that ROCloud1 and ROCloud2 achieve better application reliability under different experimental settings.

5 Related Work and Discussion

Cloud computing [4] is becoming a mainstream aspect of information technology. A number of tasks have been carried out on cloud computing, including virtualization [17], [21], [28], resource provision and monitoring [20], [43], privacy and trust [3], [10], [35] service level agreement [22], [42], storage management [40], data consistency and replication [8], [14], etc. In recent years, research investigations have been conducted on migrating legacy applications to cloud environment. [23] presented a case study of migrating enterprise IT system to IaaS cloud, which illustrated the benefits and major concerns of cloud migration. [2] surveyed various approaches for moving legacy system to SOA environment, including wrapping, replacement, etc. [46] proposed the main processes for migrating a legacy system to cloud. However, few work has been done towards improving the reliability of migrated cloud applications which is one of the major concerns during cloud migration. Complementary to the previous research efforts which were mainly focused on the procedure of migration, strategies on legacy system modernization and methods to improve the cloud platform's reliability, this paper focuses on the re-design phase during the migration and proposes an optimization framework to improve the cloud application's reliability.

The main approaches in traditional software engineering include fault prevention, fault removal [41], fault tolerance [29], [47], and fault forecasting [16], [45]. For the reason of cost consideration, software fault tolerance is often employed for critical systems. While in the cloud environment, redundant components are easier to be obtained, which makes fault tolerance a feasible solution to improve application reliability. The major techniques of software fault tolerance include recovery block [36], N-Version Programming (NVP) [6], N self-checking programming [26], distributed recovery block [24], etc. Based on these techniques, passive and active strategies are adopted by different systems: passive strategies in FT-SOAP [15] and FT-CORBA [39], while active strategies in FTWeb [38], Thema [32], WS-Replication [37], SWS [27], and Perpetual [34]. Instead of focusing on design of fault tolerant strategies, this paper aims to select the optimal fault tolerant strategies for components and to improve the application reliability. The cloud platforms also provide techniques to improve reliability, the frequently used techniques are illustrated in [1], [13], [25]. However, only depending on these techniques are not sufficient, since not all legacy applications can be re-implemented by using map-reduce while the virtual machine restart or migration based methods can introduce latency which may not be acceptable for time-constrained applications.

Component ranking is an important research problem in cloud computing [49], [50]. Inspired by Google PageRank [9] (a ranking algorithm for Web page searching) and SPARS-J [19] (a software product retrieving system for Java), a component ranking approach for cloud application, FTCloud, is presented in [48], [50] based on the intuition that components invoked frequently by other important components are more important. Different from FTCloud, component invocation frequencies as well as the prior knowledge of component reliability (e.g., failure rate, etc.) are taken into consideration in our ROCloud approach.

The ROCloud framework is proposed for design optimization during the cloud migration process, since (1) It is time consuming to dig into the logic of legacy applications and identify significant components manually. Automatically ranking components for legacy applications becomes important, which can aid the designer to optimize the application. (2) Cloud environment is highly dynamic since its resources can scale up or scale down on-demand. Therefore, the structure of application deployed in the cloud should adapt to this highly dynamic context and be reliable and robust, which makes fault tolerance of significant components necessary. (3) The high scalability feature of the cloud makes redundant components easier be obtained. Thus, software fault tolerance becomes a feasible approach to improve the application reliability. At the same time, approaches provided by the cloud platform(e.g., virtual machine restart) can also help to build reliable cloud applications. (4) The reliability properties and the invocation relations of the legacy application can be collected since the components are running on the same server or cluster.

6 Conclusion

This paper presents a reliability-based design optimization framework for migrating legacy applications to the cloud environment. The framework consists of three parts: legacy application analysis, significant component ranking and automatic optimal fault-tolerant strategy selection. Two algorithms are proposed in the ranking phase: the first ranks components for the applications where all the components can be migrated to the cloud; the second ranks components for the applications where only part of the components can be migrated to the cloud. In both algorithms, the significance value of each component is calculated based on the application structure, component invocation relationships, component failure rates, and failure impacts. A higher significance value means the component imposes higher impact on the application reliability than others. After finding the most significant components, an optimal fault-tolerant strategy can be selected automatically with respect to

the time and cost constraints. The experimental results show that ROCloud1 and ROCloud2 outperform other approaches and can greatly improve the application reliability.

In ROCloud, each component is considered as independent and the fault-tolerant strategy selection is carried out on component basis. In the future, we will study the fault tolerance of interrelated components. In addition, ROCloud uses the ratios of component failure to application failure to measure the failure impact of components. While the relationship between component failures and application failures can be complicated, more sophisticated models (e.g., Markov models, fault trees, etc.) will be investigated in the future work.

Our future work also includes: (1) considering more factors (such as data transfer, invocation latency, etc.) when computing the weights of invocations links; (2) taking the constraint factors such as cost into consideration during the ranking phase, and letting the designer know intuitively which components can make the biggest improvement while cost the least; (3)more experimental analysis on the impact of incorrect prior knowledge such as invocation frequencies and component failure rates.

ACKNOWLEDGMENTS

The work described in this paper was fully supported by the National Basic Research Program of China (973 Project No. 2011CB302603), the National Natural Science Foundation of China (Project No. 61103032, 61100078), National Key Technology R&D Program of the Ministry of Science and Technology of China (Project No. 2013BAH01B01), and the Shenzhen Basic Research Program (Project No. JCYJ20120619153834216, JC201104220300A). Xinyu Wang is the corresponding author.

REFERENCES

[1] S. Al-kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu, "VM-Flock : Virtual Machine Co-Migration for the Cloud," in *Proceedings of the 20th international symposium on High performance distributed computing*, New York, NY, USA, 2011, pp. 159–170.

[2] A. A. Almonaies, J. R. Cordy, and T. R. Dean, "Legacy system evolution towards service-oriented architecture," in *Proc. International Workshop on SOA Migration and Evolution (SOAME 2010)*, Madrid, Spain, Mar. 2001, pp. 53–62.

[3] G. Anthes, "Security in the cloud," *Commun. ACM*, vol. 53, no. 11, pp. 16–18, Nov. 2010. [Online]. Available: http://doi.acm.org/10.1145/1839676.1839683

[4] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," *Technical Report, EECS-2009-28*, EECS Department, University of California, Berkeley, 2009.

[6] A. Avizienis, "The methodology of n-version programming," *Software Fault Tolerance, M. R. Lyu (ed.), Wiley, Chichester*, pp. 23–46, 1995.

[7] V. Batagelj and A. Mrvar, "Pajek - pajek: Analysis and visualization of large networks," *Graph Drawing Software*, vol. 21, pp. 47–57, 2003.

[8] N. Bonvin, T. G. Papaioannou, and K. Aberer, "A self-organized, fault-tolerant and scalable replication scheme for cloud storage," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC'10, New York, NY, USA, 2010, pp. 205–216.

[9] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proc. 7th Int'l Conf. World Wide Web (WWW'98)*, 1998.

[10] C. Cachin, I. Keidar, and A. Shraer, "Trusting the cloud," *SIGACT News*, vol. 40, no. 2, pp. 81–86, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1556154.1556173

[11] M. Creeger, "Cloud computing: An overview," *ACM Queue*, vol. 7, no. 5, June 2009.

[12] A. P. S. de Moura, Y.-C. Lai, and A. E. Motter, "Signatures of small-world and scale-free properties in large computer programs," *Physical Review E*, vol. 68, no. 017102, 2003.

[13] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07, New York, NY, USA, 2007, pp. 205–220.

[15] C.-L. Fang, D. Liang, F. Lin, and C.-C. Lin, "Fault tolerant web services," *Journal of System Architure*, vol. 53, no. 1, pp. 21–38, 2007.

[16] S. S. Gokhale and K. S. Trivedi, "Reliability prediction and sensitivity analysis based on software architecture," in *Proc. Int'l Symp. Software Reliability Engineering. (ISSRE'02)*, 2002, pp. 64–78.

[17] F. Hao, T. V. Lakshman, S. Mukherjee, and H. Song, "Enhancing dynamic cloud-based services using network virtualization," in *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, ser. VISA '09, New York, NY, USA, 2009, pp. 37–44.

[18] D. Hyland-Wood, D. Carrington, and Y. Kaplan, "Scale-free nature of java software package, class and method collaboration graphs," in *Proc. 5th Int'l Symposium on Empirical Software Engineering*, 2005, pp. 439–446.

[19] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Trans. Software Engineering*, vol. 31, pp. 213–225, 2005.

[20] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 261–276. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629601

[21] F. Kamoun, "Virtualizing the datacenter without compromising server performance," *Ubiquity*, vol. 2009, no. 2, Aug. 2009.

[22] A. Kertesz, G. Kecskemeti, and I. Brandic, "An sla-based resource virtualization approach for on-demand service provision," in *Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, ser. VTDC '09, New York, NY, USA, 2009, pp. 27–34.

[23] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville, "Cloud Migration: A Case Study of Migrating an Enterprise IT System to IaaS," in *2010 IEEE 3rd International Conference on CLoud Computing (CLOUD)*, 2011, pp. 450 – 457.

[24] K. Kim and H. Welch, "Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Trans. Computers*, vol. 38, no. 5, pp. 626–636, May 1989.

[25] H. A. Lagar-cavilla, J. A. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. D. Lara, M. Brudno, and M. Satyanarayanan, "SnowFlock : Rapid Virtual Machine Cloning for Cloud Computing," in *Proceedings of the 4th ACM European conference on Computer systems*, New York, NY, USA, 2009, pp. 1–12.

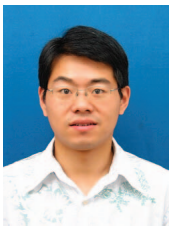
[26] J. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architectures," *Computer*, vol. 23, no. 7, pp. 39–51, Jul 1990.

[27] W. Li, J. He, Q. Ma, I.-L. Yen, F. Bastani, and R. Paul, "A framework to support survivable web services," in *Proc. 19th IEEE Int'l Symp. Parallel and Distributed Processing*, 2005, p. 93.2.

[28] X. Lu, H. Wang, J. Wang, J. Xu, and D. Li, "Internet-based virtual computing environment: Beyond the datacenter as a computer," *Future Generation Computer Systems*, vol. 29, pp. 309–322, 2013.
[29] M. R. Lyu, *Software Fault Tolerance*. Trends in Software, Wiley, 1995.
[30] M. R. Lyu, *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, 1996.
[31] P. Mell and T. Grance., "The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology," *NIST Special Publication*, vol. 800-145, 2011.
[32] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-fault-tolerant middleware forweb-service applications," in *Proc. 24th IEEE Symp. Reliable Distributed Systems (SRDS'05)*, 2005, pp. 131–142.
[33] D. Oppenheimer and D. A. Patterson, "Studying and using failure data from large-scale internet services," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, 2002, pp. 255–258.
[34] S. L. Pallemulle, H. D. Thorvaldsson, and K. J. Goldman, "Byzantine fault-tolerant web services for n-tier and service oriented architectures," in *Proc. 28th Int'l Conf. Distributed Computing Systems (ICDCS'08)*, 2008, pp. 260–268.
[35] S. Pearson, "Taking account of privacy when designing cloud computing services," in *Software Engineering Challenges of Cloud Computing, 2009. CLOUD '09. ICSE Workshop on*, may 2009, pp. 44–52.
[36] B. Randell and J. Xu, "The evolution of the recovery block concept," *Software Fault Tolerance, M. R. Lyu (ed.), Wiley, Chichester*, pp. 1–21, 1995.
[37] J. Salas, F. Perez-Sorrosal, n.-M. Marta Pati and R. Jiménez-Peris, "Ws-replication: a framework for highly available web services," in *Proc. 15th Int'l Conf. World Wide Web (WWW'06)*, 2006, pp. 357–366.
[38] G. T. Santos, L. C. Lung, and C. Montez, "FTWeb: A fault tolerant infrastructure for web services," in *Proc. 9th IEEE Int'l Conf. Enterprise Computing*, 2005, pp. 95–105.
[39] G.-W. Sheu, Y.-S. Chang, D. Liang, S.-M. Yuan, and W. Lo, "A fault-tolerant object service on corba," in *Proc. 17th Int'l Conf. Distributed Computing Systems (ICDCS'97)*, 1997, p. 393.
[40] S. Sivathanu, L. Liu, M. Yiduo, and X. Pu, "Storage management in virtualized cloud environment," in *Proc. 3rd IEEE Int'l Conf. Cloud Computing (Cloud'10)*, 2010.
[41] W.-T. Tsai, X. Zhou, Y. Chen, and X. Bai, "On testing and evaluating service-oriented software," *IEEE Computer*, vol. 41, no. 8, pp. 40–46, 2008.
[42] I. Ul Haq and E. Schikuta, "Aggregation patterns of service level agreements," in *Proceedings of the 8th International Conference on Frontiers of Information Technology*, ser. FIT'10, New York, NY, USA, 2010, pp. 40:1–40:6.
[43] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807161
[44] S. White and P. Smyth, "Algorithms for estimating relative importance in networks," in *Proc. of SIGKDD'03*, 2003, pp. 266–275.
[45] S. M. Yacoub, B. Cukic, and H. H. Ammar, "Scenario-based reliability analysis of component-based software," in *Proc. Int'l Symp. Software Reliability Engineering (ISSRE'99)*, 1999, pp. 22–31.
[46] W. Zhang, A. Berre, D. Roman, and H. Huru, "Migrating legacy applications to the service cloud," in *Proceedings of 14th conference companionon Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA09, 2009, pp. 59–68.
[47] Z. Zheng and M. R. Lyu, "A distributed replication strategy evaluation and selection framework for fault tolerant web services," in *Proc. 6th Int'l Conf. Web Services (ICWS'08)*, 2008, pp. 145–152.
[48] Z. Zheng and M. R. Lyu, "Component ranking for fault-tolerant cloud applications," *IEEE Transctions on Service Computing (TSC)*, vol. 5, no. 4, pp. 540–550, 2012.
[49] Z. Zheng, Y. Zhang, and M. R. Lyu, "CloudRank: A QoS-driven component ranking framework for cloud computing," in *Proc. Int'l Symposium Reliable Distributed Systems (SRDS'10)*, 2010.
[50] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King, "FTCloud: A ranking-based framework for fault tolerant cloud applications," in *Proc. Int'l Symposium Software Reliability Engineering (ISSRE'10)*, 2010.

Weiwei Qiu received her B.Eng. degree in College of Computer Science majoring in Computer Science and Technology from Zhejiang University in 2008. She is currently a Ph.D. candidate in the College of Computer Science and Technology, Zhejiang University. Her research interests include software engineering, software reliability, distributed computing and cloud computing.

Zibin Zheng is a postdoctoral fellow at The Chinese University of Hong Kong. He received his Ph.D. degree from The Chinese University of Hong Kong in 2011. He received ACM SIGSOFT Distinguished Paper Award at ICSE'2010, Best Student Paper Award at ICWS'2010. His research interests include service computing, cloud computing, and software reliability engineering.



Xinyu Wang is an associate professor in the College of Computer Science and Technology, Zhejiang University. He received the B.E. degree in 2002 and Ph.D. degree in 2007 from the same university. His primary research interests include Software engineering, distributed software architecture and distributed computing.

Xiaohu Yang received the B.S. degree, the M.S. degree and the Ph.D. degree all in computer science at Zhejiang University in 1988, 1990, and 1993 respectively. Currently he is a professor of Computer Science and Vice Dean of Software College at Zhejiang University in China. His research interests include software engineering, large-scale software architecture, and cloud computing.

Michael R. Lyu received the Ph.D. degree in computer science from the University of California, Los Angeles, in 1988. He is currently a Professor in the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong, China. Dr. Lyu is an IEEE Fellow and an AAAS Fellow for his contributions to software reliability engineering and software fault tolerance. Dr. Lyu is also a Croucher Senior Research Fellow.