Chinnaiah and Niranjan *Journal of Cloud Computing: Advances, Systems and Applications* (2018) 7:3
DOI 10.1186/s13677-018-0104-9

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Fault tolerant software systems using software configurations for cloud computing

Mylara Reddy Chinnaiah[1*] and Nalini Niranjan[2]

Abstract

Customizable software systems consist of a large number of different, critical, non-critical and interdependent configurations. Reliability and performance of configurable system depend on successful completion of communication or interactions among its configurations. Most of the time users of configurable systems very often use critical configurations than non-critical configurations. Failure of critical configurations will have severe impact on system reliability and performance. We can overcome this problem by identifying critical configurations that play a vital role, then provide a suitable fault tolerant candidate to each critical configuration. In this article we have proposed an algorithm that identifies optimal fault tolerant candidate for every critical configuration of a software system. We have also proposed two schemes to classify configurations into critical and non-critical configurations based on: 1) Frequency of configuration interactions *(IFrFT)*, 2) Characteristics and frequency of interactions *(ChIFrFT)*. These schemes have played very important role in achieving reliability and fault tolerance of a software system in a cost effective manner. The percentage of successful interactions of *IFrFT* and *ChIFrFT* are 25 and 40% higher than that of the *NoFT* scheme. In *NoFT* scheme none of the configurations are supported by fault tolerance candidates. Performance of IFrFT, ChIFrFT, and NoFT schemes are tested using a file structure system.

Keywords: Configurable software systems, Fault tolerance, Reliability, Configurations interactions

Introduction

Customization of a software system varies with user requirements or target platform. Programmers employ preprocessor directives, command-line arguments, setup files, configuration files to customize a software system. Using product-line technology, it is possible to generate a program tailored to individual user requirements by using program generators. Program generation process leverage software system features where a feature is a *visible behavior or characteristic* of a software program [1]. According to product-line technology, any customizable option that can be selected during the compile or load time is called a *feature* of a program. Program generator generates a program depending on features selected by the user(s).

Every program or software system will have functional and non-functional properties. Functional properties are activities that are to be performed or a behavior that has to be exhibited by a program when a specified condition is met. Most of the non-functional properties are related to performance, such as accessibility, fault-tolerance, reliability, scalability, recoverability, maintainability and availability of program(s). Most of the time users will be interested in non-functional properties such as allocation and release of memory, emailing electronic advertisements to users. A *software configuration* consists of non-functional properties customized to a specific set of features. For example, preprocessor directives customized to a particular set of requirements and processed during compilation of software system will become configurations.

Generally software systems consists of several different configurations, in turn, each configuration consist of many features. Similar to a configuration model with a few configurations produce several variants of *software systems* in [2, 3], a feature model that consists of few features can produce several numbers of configurations. An *interaction* is a communication between two or more configurations of a software system to exchange data. A software system that consists of hundreds or thousands of

*Correspondence: mylarareddy@revainstitution.org
[1]School of Computing and Information Technology, REVA University, Rukmini Knowledge Park, Yelahanka, 560064 Bangalore, India
Full list of author information is available at the end of the article

configurations having different failure behavior is prone to failure.

Configurations that perform important, common or default operations are called *critical configurations*. During the execution of software system many non-critical configurations interact with critical configurations. Critical configurations are specialized versions of non-critical configurations similar to many features are extended versions of important features [4].

A configuration is said to be *failed configuration* if it either produce errors during its execution or fail to successfully complete its task. A software system that consists of configurations which are prone to failure will have unpredictable behavior and performance anomalies making it unusable or untrustworthy. Communication (parameters, return values, etc.) between two or more configurations is called *Configurations Interaction*. Incomplete/failed communication between two or more configurations is said to be *Failed Interaction*. The significance level of critical configuration will come down when it is involved in failed interactions. Failed interactions have become common than exceptions [5] in modern complex software systems. Further, by classifying the configurations into two categories as given below, we can improve software systems reliability and fault tolerance.

1) Frequently used or Critical configurations, and

2) Less frequently used or non-critical configurations

After the classification of configurations into the categories mentioned above,frequently used configurations are backed up by fault tolerant candidates to improve the reliability and performance of software systems.

According to software reliability engineering, the main approaches to build reliable software systems are 1) Fault Forecasting [6, 7], 2) Fault Prevention, 3) Fault Removal [8] and 4) Fault Tolerance [9]. Fault prevention and fault tolerance techniques are leveraged in the development of large and reliable complex software systems. To tolerate faults, design diversity technique proposed by Avizienis et al. [10], employs functionally equivalent yet independently designed and developed software system modules. Since this technique incur high development and maintenance cost it is employed only in the development and maintenance of machine critical systems (disaster control, threat of loss of life).

Software systems usually consist of hundreds of configurations. Although provisioning redundant configurations for every configuration of large and complex software system help to tolerate failures, it would be very expensive and results in high upfront investment cost. It is possible to minimize investment cost by provisioning redundant configurations only for frequently configurations. Microsoft has reported that it has reduced 80% of failures by fixing top 20% of most frequently reported bugs or crashes in its windows and office suite. Generally 80% of end users use only 20% of software application features [11].

Our contributions in this paper are as follows:

- Classification of configurations of software systems into following categories: 1) Frequently used (Critical) configurations and 2) Less frequently used (non-critical) configurations.
- Compare the performance of the following proposed strategies 1) Frequently-used (Critical) configurations enabled with fault tolerance, 2) All configurations of a software system enabled with fault tolerance, 3) None of the configurations have fault tolerance support.
- Demonstrate practicality of the proposed strategies on a file structures system that consists of several configurations.

Organization of this paper is as follows: "Related work" section presents the related work, "System model and failure model" section present system model and failure models used in this work, "Overview of proposed approach" section introduces an overview of the proposed approach and system architecture, "Classification of configurations" section presents a mathematical model to classify configurations, "Fault tolerance schemes" section presents various fault tolerant schemes, selection of most suitable fault tolerant candidate to each configuration, finally the "Experimental setup and performance evaluation" section presents experimental setup and results discussion.

Related work

In cloud computing environment occurrence of failures is random in nature and also there may be unknown types of failures. It is important either to eliminate failures permanently or minimize the impact of failures. Fault tolerant approaches are broadly classified into two categories. 1) Reactive schemes and 2) Proactive schemes.

Reactive schemes swing into action as soon as software system failed. Some of the popular reactive techniques are:

a) Checkpoint-Restart: In this approach the overall execution time of a job/task depends on checkpoint intervals. Shorter checkpoint intervals cause frequent checkpointing, large number of checkpoints thus resulting in much time spent in checkpoint activity, higher disk space required to store checkpoint images. Longer checkpoint intervals cause high resource wastage due to re-execution of a job from its longer distant previous checkpoint. A checkpoint interval that is very good for some workloads may be worse for other workloads. For example, in case of transmission failures shorter checkpoint intervals completely eliminate retransmission of lengthy messages of TCP applications (deal with longer messages) while

checkpointing UDP (deal with short messages) applications with longer intervals eliminate the need for large storage space to save checkpoint images and reduce the cumulative compute power overhead of all checkpoints. Research on finding most suitable checkpoint intervals for different workloads, yet to happen. Checkpointing identical operations performed by many users on several computing machines will lead to wastage of resources and increased cost. To overcome this problem a technique proposed by Zhou et al. in [12] checkpoint identical parts of all virtual machines which host and render identical services.

b) Job migration: Job migration can be further classified into preemptive and non-preemptive migration. In preemptive migration mechanism a job undergoes migration whenever a monitoring system detects misbehavior of the job. Accuracy of preemptive migration schemes depends on fitness of failure prediction algorithms. Incorrect predictions lead to unnecessary job migrations, increased resource wastage and longer job turnaround time. In this approach Mean Time To Migrate (MTTM) consists of time taken i) by prediction algorithm, ii) to notify run time environment iii) to complete the migration task and iv) how best the prediction algorithms know types of failures that occur. In preemptive approach Mean time to migrate a job is less compare to non-preemptive approach. Preemptive approach is more suitable for critical machine control systems such as aviation, nuclear control systems, etc.

c) Replication: In replication mechanism a job is replicated over multiple compute platforms in which replicas are executed in parallel. Voting mechanism is used in case of more than two replicas to decide correct output. As an alternative method to parallel execution of replicas, the concept of primary and secondary was proposed. In this approach secondary job takes over the role of primary as soon as primary job failed. Byzantine Fault Tolerance (BFT) approach [13] proposed by Zhang et al. make use of $3k+1$ replicas to overcome K faults. In gossip approach proposed by Lim et al. [14] $2K+1$ replicas are used to tolerate K faults. For further reading about job migration techniques interested reader may refer following articles [15–19].

To improve the fault tolerance of distributed applications in a cloud computing environment, Zhao et al. in [20] proposed a fault tolerant middleware that consist of three components: 1) A low level messaging protocol that render reliable, totally ordered multicast service between primary and backup members of a group. 2) A leader-determined membership protocol to ensure that every member of a group has a consistent view of primary member and other members belong to its group, also shorten the time required to elect a primary member among several members of the group. 3) A virtual determiner framework that transforms every non-deterministic operation of a primary member to virtually deterministic operation while guaranteeing that all backups shall receive results in the same order as that of the primary member.

Proactive schemes: In proactive schemes hardware and computing environment continuously monitored for occurrence of failures by employing best failure prediction algorithms. Prediction algorithms take decisions based on the outcome of intermittent results and communication messages. Whenever a monitoring system detects deviation in behavior of a job due to failure of underlying hardware or computing environment, then the job is migrated to fault free computing environment. Some of the schemes in proactive fault tolerance are as given below:

a) Self Healing: Self healing systems detect and respond to malfunctions that occur during run time. They have the ability to recognize failures and take suitable action immediately to minimize the impact of failures. Functional and interaction failures are common that occur during run-time in component based software systems. An approach proposed by Nicolo P. [22] exploits redundancy of components to mask failures. Cliff Saran in [23] highlighted some of the difficulties involved in usage of self-managing systems such as common software standards across the industry and proprietary system management interfaces. Quality of service of an application is ensured by dealing with its quality attributes. In [24] V. Nallur et al. have proposed a self-optimizing architecture based on service level agreements. Using this architecture any service hosted on the cloud will dynamically self-manage depending on service provider inputs to minimize the cost of service. As far as possible this architecture ensures that the user service level agreements are satisfied while the service is being re-configured according to service provider input.

Execution of parallel applications in a cloud computing environment requires multiple processing nodes. Processing nodes that have poor coordination/synchronization will produce incorrect results and would lead to wastage of resources. An algorithm proposed in [25] schedule parallel applications based on virtual machine characteristics to reduce consumption of physical resources such as network elements and power within the data center.

b) Software Rejuvenation: This approach deal with handling transient failures caused by software aging phenomenon. It involves resetting a part of the internal state of a job followed by restarting the execution of a job. Fault tolerant software systems with two-version redundant structures and single-version rejuvenation were proposed in [26] and [27] respectively. An approach based on backup of virtual machines in the cloud [28] was proposed by Xinyi et al. to improve system reliability. For more

information on software rejuvenation interested reader may refer to [29–32].

c) Preemptive Migration: Migration of processes, operating systems and virtual machines enable load balancing, efficient resource usage, fault management and system maintenance. Migration of virtual machines, computing platform across homogeneous and heterogeneous hosts help administrators manage data centers, clusters in easy and efficient manner. Some of the research articles published in this area are [33–36]. Proactive recovery techniques such as staggered proactive recovery, rebooting a compromised node, refreshing the state of a node do not guarantee hundred percent recovery of nodes and are not sufficient enough to prevent Byzantine faults permanently. Since the reboot and recovery operations are performed as part of the critical execution sequence, they take more time to achieve a stable state by recovering failed node. A proactive recovery mechanism based on service migration [21] was proposed to overcome limitations of reboot and recovery mechanisms. In this scheme pool of spare nodes are used to migrate long-running applications off the critical path, run as background/separate tasks, thereby minimizing the reboot and recovery time.

Fault tolerance can be considered during the design, development and architecture of large, complex software systems [6]. Considering virtual machines and cloud elasticity property Xiaomin Zhu et al. have proposed an algorithm, FASTER, [37] that overlap workload tasks to improve reliability of scientific workloads in virtual clouds. Seigmund et al. in [38] have proposed an approach that treats customizable systems as a black box and detect performance relevant feature interactions on a set of specially selected configurations. The pair-wise heuristic technique was used to and measure a set of configurations that select cover all pairwise interactions among configurations relevant to performance.

Liebig et al. in [39] have analyzed the variability characteristic in forty software programs ranging from small to large scale and claimed that structural interactions occur betweens two features and such interactions in turn may cause performance feature interactions. Sincero et al. in [40] proposed a model that deals with feature attributes, interaction among features that predict configurations non-functional properties. Software feature behavior that produce an unexpected outcome indicates malfunctioning of software feature. Zheng et al. [41] and Slawinska et al. [42] have proposed collaborative solutions for specific applications such as MPI. Parnas et al. [43] and Shooman et al. [44] highlighted the impact of the structure of an application on its performance, reliability and correctness.

Fault tolerance techniques proposed in this article are based on replication. As there exist many robust, efficient, optimal fault tolerant schemes in the literature related to hardware, middleware, platform levels, we assume that there are no failures in hardware, middleware, and platform levels. Fault tolerance in hardware, middleware, computing platform is out of the scope of this article. In this article we address fault tolerance at the application level by exploiting application characteristics such as structure and feature's behavior. In this direction we propose a scheme to identify frequently used configurations which play vital role in a software system, then enable them configurations using suitable fault tolerance candidate to improve reliability and fault tolerance of a software system. Also, we investigate the importance of configurations interactions on system performance, reliability and fault tolerance.

System model and failure model

- System Model: The various operations of file structures techniques involve interaction among

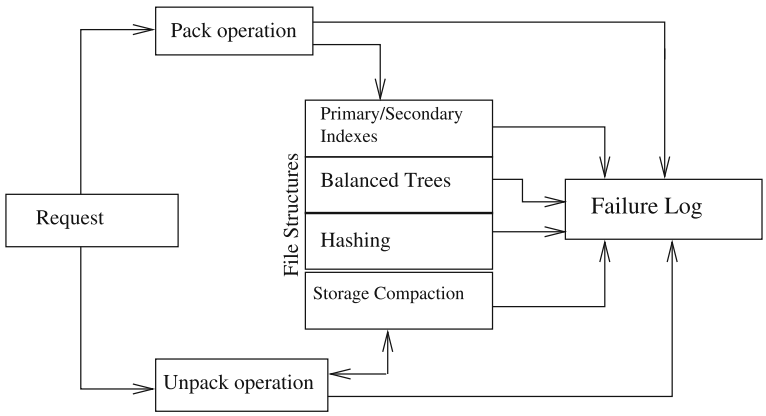

Fig. 1 System Model

several configurations and successful execution of their interactions. Pictorial representation of the system is shown in Fig. 1. File structure system has several configurations, each of which have their functionally equivalent yet an independent configuration written in different programming logic. Every independent configuration act as fault tolerant candidate. Fault tolerance of the system depends on the number of redundant configurations, type of failures and number of failures. In our file structure system each critical configuration is supported by only one fault tolerant candidate and two types of failures are dealt with. Requests are generated to save new data records, fetch existing data records from the file structure. Pack operation is invoked by all techniques of file structures for every request to create a new record. Similarly unpack operation is invoked for every request to fetch data record(s) from file structures.

Value of record fields are passed to/returned from one configuration to another configuration. During this process faults are introduced by flipping some bits of the fields of a record through random selection. When faults are introduced in non-critical configurations, then such configurations are treated as permanently failed configurations. Subsequent requests for such configurations are considered as unsuccessful/failed interactions. When faults are introduced in a critical configuration that has the support of the fault tolerant candidate, then subsequent requests will be delegated to the fault tolerant candidate. However, upon failure of fault tolerant candidate subsequent requests targeted to it are treated as failed requests.

- Failure Model: In the file structure system we have considered 1) Response failures: a) Value failure, b) State transition failure and 2) Omission failures: a) Receive omission failure, b) Send omission failure as given in Table 1. Value failures and state transition failures are injected randomly into the record fields to flip values of some of the bits of record fields. Value failures are injected into the parameters and return value of functions. State transition failures occur when we change the value of variable(s) used in conditional constructs resulting in execution of different part of a program. For example block of statements belong to the else part of an if statement instead of true part. Omission failures occur during read and write operations on file structure. During the execution of a request to create a new record some of the bits of record fields are set and reset to introduce failures and such failures contribute to receive omission failures. Similarly, while reading record fields some of the record fields are modified by setting and resetting bits and such failures are called send omission failures. As shown in Table 1 response failures occur in pack, indexes, balanced and hashing techniques while omission failures are generated in unpack, indexes, balance trees, hashing, and storage compaction techniques. For example hash key is modified while a record is being hashed. As a result of modified key, hash algorithm may or may not produce collisions, that is, it might produce collision for a key to which collision should not occur and vice-versa.

Table 1 Failure models

Type of failure	Operations/techniques
Response failures	Pack, Indexes, Balanced Trees, Hashing
1. Value failure	
2. State transition failure	
Omission failure	Unpack, Indexes, Balance Trees, Hashing, Storage compaction
1. Receive omission	
2. Send omission	

Overview of proposed approach

Figure 2 depicts a pictorial representation of the proposed architecture to address some of the challenges mentioned above. It consists of two parts: 1) Classification of configurations and 2) Selection of the most suitable, optimal fault tolerant candidate to every critical configuration. Procedure to achieve fault tolerance of a software system is as follows:

- We have used two versions of the file structure software system. In first one, all configurations of a software system are supported by their replicas. In the second version, only critical or frequently used configurations, which contribute to 20% of all configurations, are supported by suitable fault tolerant candidates that are functionally equivalent yet independent configurations developed using different programming constructs/logic to mask failures.
- After the completion of all interactions, interaction value of each configuration is calculated based on the number of successful interactions.
- Configurations are classified into most frequently used (critical) and less frequently used (non-critical) categories based on their interaction values.
- Effectiveness of each fault tolerance strategy is measured and compared with other strategies, then most suitable fault tolerance strategy for a given critical configuration is selected.

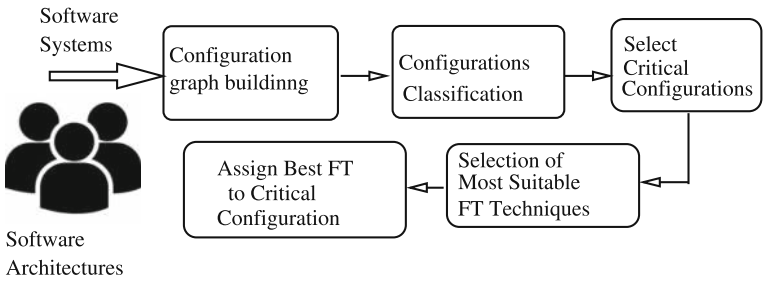

Fig. 2 System architecture of the proposed work

Classification of configurations

Configurations interaction graph building

A software system can be modeled as a weighted directed graph denoted by G, where C_i represent i^{th} configuration and an edge denoted by I_{jk} represent interaction between configurations C_j and C_k, that is, configuration C_j invokes C_k. In this process every configuration, say C_i, that is involved in successful interaction has a non-negative interaction value $P(C_i)$. At the end of all interactions the percentage of successfully executed interactions between every pair of configurations, for example, between C_j and C_k, are calculated using Eq. 1

$$M\left(I_{jk}\right) = \frac{F_{jk}}{\sum_{k=1}^{N} F_{jK}} \tag{1}$$

Where F_{jk} represent the number of times configuration C_j invoked C_k and N is the number of configurations present in a software system. If there is no interaction between C_j and C_k then $F_{jk} = 0$. Value of I_{jk} increased by *one* for every successful interaction between C_j and C_k. An interaction I_{jk} will have a larger value if C_j interacts with C_k more number of times than other configurations. For a software system having N configurations, the configuration graph represent $N * N$ matrix, denoted by M, that can be obtained from Eq. 1. Each element of M, denoted by m_{jk}, represent interactions between j^{th} and k^{th} configurations. A configuration interaction value between every pair of configurations is calculated according to the steps given below:

- If j^{th} configuration during its life time never interact with k^{th} configuration, then the interaction value of j^{th} configuration with k^{th} configuration will be *zero*, that is, $M\left(I_{jk}\right).m_{jk} = 0$
- If j^{th} configuration interacts with itself, in case of recursive configurations, then its interaction value represented by m_{jj} is incremented by *one* for every successful interaction.
- If j^{th} configuration interact with none of the configurations except k^{th}, then the interaction value will be calculated as $I_{jk} = \frac{1}{N}$ for all $k = 1$ to N except j. Matrix M is a stochastic matrix since all the elements in a row of M sum to *one* and each element lies in the range [0, 1].

Configurations classification

Software system configurations are categorized based on :

1) Frequency of interactions between configurations (*IFrFT*)

2) Characteristics such as structure (critical or non-critical) and frequency of interactions (*ChIFrFT*)

IFrFT: fault tolerance based on frequency of interactions

In this approach all configurations are treated as equal by setting their individual interaction value to *zero* at the beginning of execution of a software system. Configurations that are involved in most of the interactions during execution of software system are considered as critical configurations. Reliability and performance of any software system depend on successful execution of its critical configurations. For an increased number of failed critical configurations and configuration interactions reliability and performance of software system deteriorate significantly. By employing fault tolerant candidates to critical configurations, successful execution of most of interactions, it is possible to improve the reliability and performance of a software system. In this context, we propose a mathematical model to find frequently used configurations of a software system based on frequency of interactions. The proposed mathematical model is as follows:

- Configuration C_i is assigned to a value $\frac{V_i}{\sum_{j=1}^{N} V_j}$, where N is the number of configurations, V_i is the significance of i^{th} configuration.
- Compute the interaction value of i^{th} configuration C_i, denoted by $P\left(C_i\right)$, for $i = 1$ to N, using

$$P\left(C_i\right) = \frac{1-\alpha}{N} + \alpha \sum_{j \in S(C_i)} P(C_j) M\left(I_{ji}\right) \tag{2}$$

where $S\left(C_i\right)$ is a set of configurations that invoke interactions with configuration C_i. Variable α $(0 \leq \alpha \leq 1)$ in Eq. 2, indicate the interactions

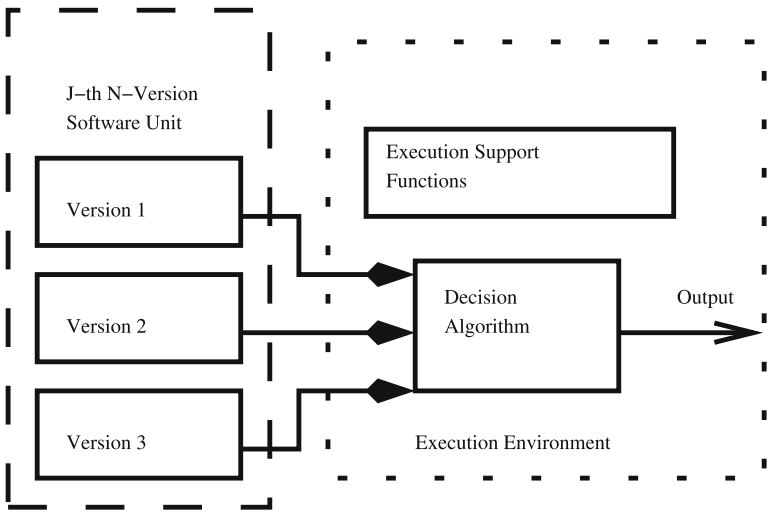

Fig. 3 N-Version programming

influence, in terms of interaction value, between C_i and C_j. Therefore, an interaction value of C_i is derived from configurations that have invoked it for interaction. Interaction value of C_i depends on $|S(C_i)|$, $P\left(C_j\right)$, $M\left(I_{ji}\right)$. The larger value of C_i indicates that it has been invoked very frequently by other configurations. Using Eq. 2 we find frequently used configurations of a software system. Value of α is varied between *zero* and *one* until we get stabilized interaction values for 10000 interactions.

Using Eq. 2 we obtain configurations that are frequently used and play important role in improving the reliability of a software system.

ChIFrFT: fault tolerance based on characteristics and interaction frequencies of configurations

In this approach software configurations are categorized into two sets 1) A set, denoted by *C*, of configurations which perform critical tasks (checking for uniqueness, size of index table, search operation, and collisions are treated as characteristics) 2) A set, denoted by *UC*, of configurations which perform non-critical tasks (the padding of special symbol in fixed-length records). Initial interaction values of configurations that belong to the set *C* are assigned to higher values compare to configurations of *UC*. Since the reliability and performance of software systems depend on successful execution of their critical configurations, we can improve the reliability, performance by provisioning fault tolerant candidates to critical configurations.

Performance (defined in terms of interaction value and the number of successful interactions) of a critical configuration, say C_i, where $C_i \in C$ is computed by using Eq. 3 given below:

$$P\left(C_i\right) = (1-\alpha)\frac{\beta}{|C|} + \alpha \sum_{j \in S(C_i)} P\left(C_j\right) M\left(I_{ji}\right) \tag{3}$$

For a non-critical configuration C_k, where $C_k \in NC$, we compute the performance using Eq. 4 given below:

$$P\left(C_k\right) = (1-\alpha)\frac{1-\beta}{|NC|} + \alpha \sum_{j \in S(C_k)} P\left(C_j\right) M\left(I_{jk}\right) \tag{4}$$

Where $S\left(C_k\right)$ is the set of configurations that invoke configuration C_k, sum of $|C|$ and $|NC|$ is equal to N. In *ChIFrFT* approach the parameter β is used to find the importance of configurations belong to sets *C* and *UC*. For $\beta \leq \frac{|C|}{|NC|}$ performance of *ChIFrFT* approach is same as that of *IFrFT*, that is, frequently used and less frequently used configurations are treated equally. However, *ChIFrFT* performs better than *IFrFT* for $\frac{|C|}{|NC|} \leq \beta \leq 1$, that is, frequently used configurations are treated with higher priority than less frequently used configurations.

Fault tolerance schemes

Classification of software fault tolerance techniques

Software fault tolerance schemes are classified into two major categories. 1) Proactive schemes and 2) Reactive schemes

- **Proactive Fault Tolerance Schemes:** Proactive fault tolerance schemes are adopted in computing systems which suffer severely due to failures. Proactive schemes are costlier as they need lots of resources to support simultaneous execution of primary job and its replicas. Some of the proactive fault tolerance schemes are:

  1. **Preemptive migration:** Preemptive migration [45] techniques require very good failure

prediction algorithms. For instance prediction of faults related to performance, resource outages and functional failures for timely actions. Effectiveness of preemptive migration schemes depends on the correctness of the output of prediction algorithms.

2. **N-Version programming:** In N-version programming [46] N programs, $N \geq 2$, that are functionally equivalent yet independent as shown in Fig. 3 are generated from the initial formal specification defined using any specification language. Some of the specification languages are *Clear, OBJ3, CafeOBJ, ACT, ASL, ASL+, SPECTRUM*, and *Larch*.
3. **N-Copy programming:** N-Copy programming as shown in Fig. 4 is the data diverse complement of N-Version Programming. All copies of programs execute in parallel by taking data produced by data re-expression algorithms [47].
4. **Byzantine fault tolerance:** A Byzantine fault is an incorrect operation that occurs in a distributed environment. Byzantine faults can be classified into 1) Omission failure: Failure of a resource means requested resource might not exist or unavailable due to busy 2) Execution failure: Failure due to sending incorrect or inconsistent data, corrupting local state or responding with incorrect data, for example, round-off errors propagated from one function to another function [48].
   To provide Byzantine fault tolerance in presence of m faulty processors, (1) There must be at least $3m + 1$ processors, (2) Must exist, at least $2m + 1$ communication paths between a pair of processors, (3) There must be $m + 1$ rounds of messages exchanged and (4) Processors must be synchronized within a known skew of each other.

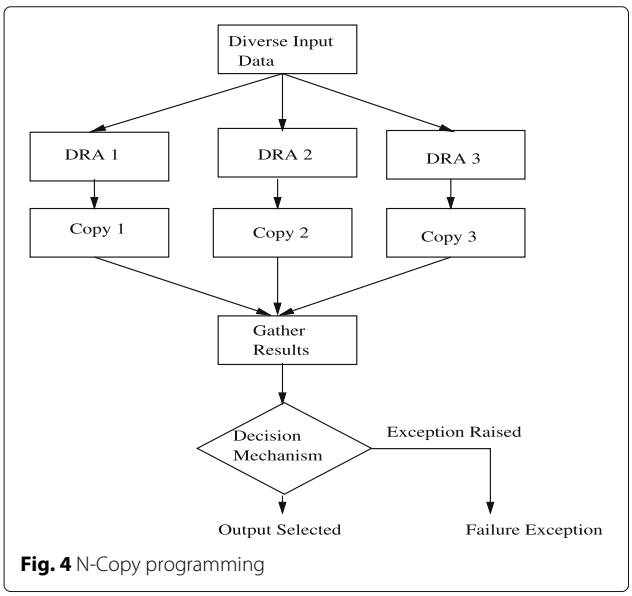

Fig. 4 N-Copy programming

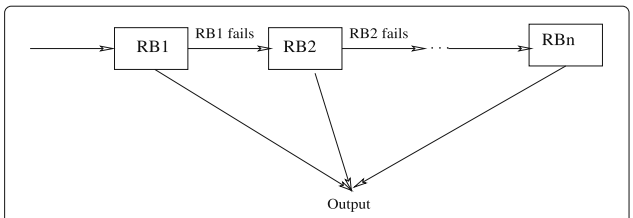

Fig. 5 Recovery Block programming

- **Reactive Fault Tolerance Schemes:** Reactive fault tolerance schemes act on the recovery process after the occurrence of failure. These schemes take less amount of resources, more time for recovery, compare to proactive schemes. Some of the reactive schemes are:

  1. **Recovery Block programming:** Recovery block [49] is a popular technique employed in software fault tolerance domain. Figure 5 depicts the pictorial representation of recovery block technique. This technique makes use of redundant program modules. Redundant program modules are executed in sequential manner. For example, in Fig. 5 RB_2 executed when RB_1 failed, RB_3 executed when both RB_1 and RB_2 failed. Similarly redundant block RB_n is executed when all the redundant blocks other than RB_n, that is RB_1 to RB_{n-1}, are failed.
     This technique fails when all redundant program modules failed. Probability of failure of recovery block technique, denoted by F, can be calculated by using the equation given below:

     $$F = \prod_{i=1}^{n} f_i \tag{5}$$

  2. **Process checkpointing:** Checkpointing is a technique that is commonly used to reduce execution time of long running programs in the presence of failures [50]. In this technique, the status of the program/job under execution is saved intermittently. Program/job resumes its execution from the most recent checkpoint instead of from the beginning of the program/job upon occurrence of failure. For a program/job,

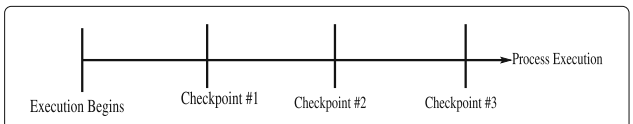

Fig. 6 Checkpoint Technique

Table 2 Overview of software system considered

System	Domain	Language	LOC	Configurations
File structures	File handling	C++	2136	92

having n modules with n-1 equally spaced checkpoints, as shown in Fig. 6, expected execution time of a program/job is given by,

$$E\left(T\left(_{x,n}\right)\right) = \left(\frac{1}{\gamma} + E(R)\right)\left[(n-1)\left(\phi_c\left(-\gamma\right)e^{\frac{\gamma x}{n}} - 1\right) + \left(e^{\frac{\gamma x}{n}} - 1\right)\right] \tag{6}$$

where x is program/job processing requirement, random variable C is a checkpoint duration, R is a random variable that represents repair time, γ is the rate at which failures occur according to a Poisson distribution, ϕ_c is the Laplace-Stieltjes transformation (LST), $E(R)$ is the expected repair time.

3. **Process migration:** It is an act of transferring state information of live processes from one machine to another across homogeneous or heterogeneous hardware, platforms [51]. Process migration technique to realize its full potential must be able to perform the following tasks 1) A process must preserve its internal state during migration, 2) Must be efficient and 3) Must be as transparent as possible to the external environment.

Selection of best fault tolerance scheme

Each fault tolerant scheme will have several unique candidates based on their configurations and characteristics such as response time, resources like computing power, storage, etc. For example, replication fault tolerance schemes are storage and compute power intensive whereas process migration schemes are time intensive. A fault tolerance scheme, say F_i, may have J candidates F_{ij} for $J = 1$ to n. Given the interaction value of a configuration, values of all eligible fault tolerant schemes (set of redundant configurations which are functionally equivalent, but independent) that are suitable to C_t as arguments to Algorithm 1, it selects the most optimal fault tolerant candidate from the set of eligible fault tolerant candidates. Each fault tolerant scheme will have several candidates depending on the characteristics and constraints to be satisfied. To select the most suitable and optimal fault tolerant scheme to a given critical configuration above algorithm uses failure probability of every fault tolerant scheme. For the set of constraints of a critical

Algorithm 1 Select Best Fault Tolerance Scheme

Input: Array of FT candidate values, value of a configuration that require FT

Output: Best FT candidate for the given configuration

```
1:  function SELECT_BEST_FT_CANDIDATE(t[i], C_tk) ▷
    Where t[i] - FT values, C_tk is an interaction value
2:      x = 0
3:      for i = 1 to n do
4:          if t[i] <= C_tk then
5:              S[x] = t[i]
6:              x = x + 1
7:          end if
8:      end for
9:      min = S[1]
10:     for j = 2 to x do
11:         if min > S[j] then
12:             min = S[j]
13:             FT(C_t) = j
14:         end if
15:     end for
16: end function
```

```
int Student :: Pack(FixedLengthRecord & Buffer) const
{
//pack the fields into a FixedLengthRecord,
// return TRUE if all succeed, FALSE otherwise

   int result;
  result =Buffer.Pack(Regno);
   result = result && Buffer. Pack(FirstName);
    ---
    ---
    ---
  return result;
}


int Student :: Unpack(FixedLengthRecord & Buffer)
{
    int result;
    result = Buffer.Unpack(RegNo);
    result = result && Buffer.Unpack(FirstName);
    ---
    ---
    ---
   return result;  // if result is 0 then causes null value error
  }
```

```
int IndexofRecords (char *filename, TextIndex      & RecordsIndex)
 {
    Record rec; int addr, result;
    DelimitedFieldBuffer Buf;
    res = RecordIndex . open(filename, ios::in);
  if(! res)   {
                cout << " File Opening Error";
                return 0;
               }
  addr = RecordIndex . Read();
  if(addr < 0)
  cout<< "Invalid Record Address";    //Incorrect index record address
     .
     .
     .
 }
```

Fig. 7 Code snippet for pack, unpack and index operations

Table 3 Possible failures in each operation

Operation	Role and specification	Possible failures
Pack operation on fixed and variable length records	Concatenation of attributes of fields to compose a record	1. Overflow of record fields 2. Data of a record exceeds the length of the record 3. Null values to record fields 4. Data type mismatch between record fields and values 5. File cannot be opened for write operation
Unpack operation on fixed and variable length records	Extraction of attributes of fields of a record	1. Null value of a field 2. Type mismatch between target data type and field value 3. File not found 4. File cannot be opened for read operation
Search using relative record number (RRN)	Search an index file for a record whose RRN is same as a search key	1. Record not found 2. Invalid RRN 3. RRN exceeds maximum value
Primary index and Secondary index	Building primary and secondary indexes for a data file	1. Primary key constraint violation 2. Incorrect index record address 3. Invalid secondary key value
Balanced trees	Balanced trees to manipulate data records	1. Incorrect leaf node 2. Incorrect key index 3. Deletion of a key from empty node
Hashing	Generate a unique address to each data record to store it in a hash table	1. Multiple keys hash to the same address 2. Overflow of hash table
Storage compaction	Method to reuse memory reserved for data records which no more exist	1. Internal and external fragmentation

configuration given as input to the algorithm, it generates a list of suitable fault tolerant candidates whose failure probability is less than the configuration interaction value. Finally the best FT candidate that has minimum failure probability among all possible candidates will be assigned to the critical configuration.

Experimental setup and performance evaluation

Using C++ language we have developed a software system for file structure techniques proposed by Michael J. Folk in [54]. Details of the software system are given in Table 2. It consists of several configurations which are functionally equivalent yet independent to achieve fault tolerance. The file structure system involves following operations: *1. Fixed-length record packing 2. Variable-length record packing 3. Unpacking of fixed-length records 4. Unpacking of variable-length records 5. Searching for a record using Relative Record Number(RRN) 6. Create Primary Index 7. Create Secondary Index 8. Balanced Trees 9. Hashing 10. Dynamic Hashing and 11. Storage Compaction*

Figure 7 shows a sample code snippet for record packing, unpacking, and index operations.

Table 3 shows the list of various operations, role and specifications, possible failures of each operation.

Table 4 shows the list of various failures caused by failed interactions and their descriptions.

We have used two versions of a file structure system 1) With all of its configurations enabled, 2) Functionally equivalent yet independent versions of frequently used (critical) configurations that are 1%, 5%, 10%, 15%, and 20% of all configurations which act as a fault tolerant plane. The file structure system is deployed on VMWare vCenter Server 4.1 that runs on PowerEdge T340 having following configuration: Intel Xeon E5-2430, 2.2 GHz, 16 GB RAM and vSphere Cleint 4.1 installed on the remote desktop computer. Execution requests to multiple groups that consists of both frequently used and less frequently used configurations are generated using a synthetic workload generator. Every execution request randomly generates values for fields of a data record. The operations listed in Table 3

Table 4 List of possible failures and their description

Failure name	Description
Overflow	Occur when size of data exceeds the maximum length of a field or record
Null value	Occur when nothing is set to record field
Invalid data	Occur when there is a type mismatch between data type of a field and the value being assigned
Read, Write	Occur when a specified file cannot be opened for read, write operations
Duplicate values	Occur when a field has similar attributes
Invalid RRN	When a given RRN exceeds its maximum value or larger
	than the current largest RRN
Primary key constraint violation	Duplicate key exist in primary index
Collision	When two or more keys hash to the same address of hash table
Invalid field value	When there is a type mismatch between data type of a field and the value

cause interactions between dependent configurations. By monitoring configuration interactions for failure, information about failed interactions logged for further processing.

Fault tolerance techniques *IFrFT*, *ChIFrFT*, *AllFT*, *NoFT* and *FT candidate selection algorithm* are implemented in C++ language. In *IFrFT* and *ChIFrFT* techniques frequently used configurations are supported by fault tolerant candidates. None of the configurations are supported by fault tolerant candidates in *NoFT* technique. In *AllFT* technique every configuration is supported by their replicas. The percentage of failed configurations varied from 10 to 100% in steps of 10. The maximum number of frequently used configurations is set to 20% of all configurations. Value for parameter α in Eq. 2 varied between 0.7 and 0.9 to identify the best value as it is used in [52, 53].

- Among all the schemes performance of *AllFT* is better since every configuration is supported by its replica. In this scheme replicated configurations help the system to overcome most of the failures for less number of interaction requests, for e.g. 2000 and 4000.
- Performance of *ChIFrFT* is better than *IFrFT* and *NoFT* in all the experimental scenarios since it considers both frequency of interactions and characteristics of configurations. This signifies that both characteristics and frequency of interactions of configurations play important role in improving performance and reliability of software systems.
- Reliability and performance of software systems deteriorate as the number of failed interactions increased. This effect can be nullified by provisioning fault tolerant candidates to frequently used configurations.
- For any failure percentage between 10 and 100, proposed *IFrFT* and *ChIFrFT* techniques perform better than *NoFT* while *ChIFrFT* outperform *IFrFT*.
- In case of lesser number of interaction requests, the software system has better performance and reliability when frequently used configurations that are enabled with fault tolerant candidates increased from 1 to 20%. However, performance and reliability of software system worsen for a large number of interaction requests.

We study the impact of a K percent of critical or frequently used configurations on system performance, reliability for different values of K (that is, 1%, 5%, 10%, 15%, and 20%) and interaction requests (2000, 4000, 6000, 8000, and 10000).

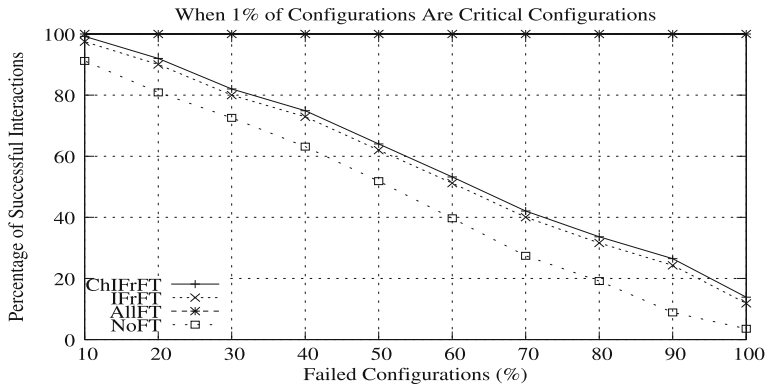

Fig. 8 Critical configurations are 1% of all configurations

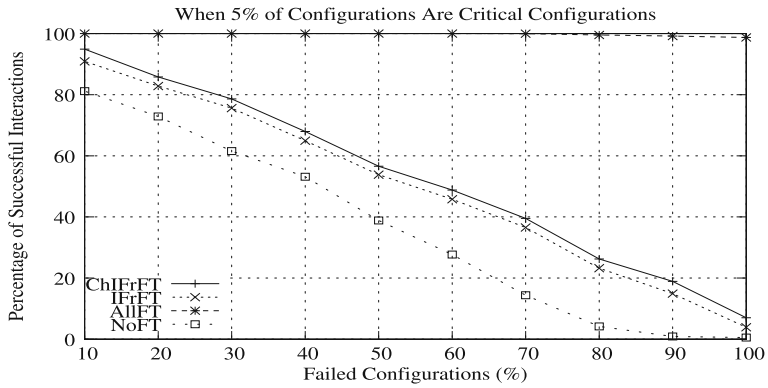

Fig. 9 Critical configurations are 5% of all configurations

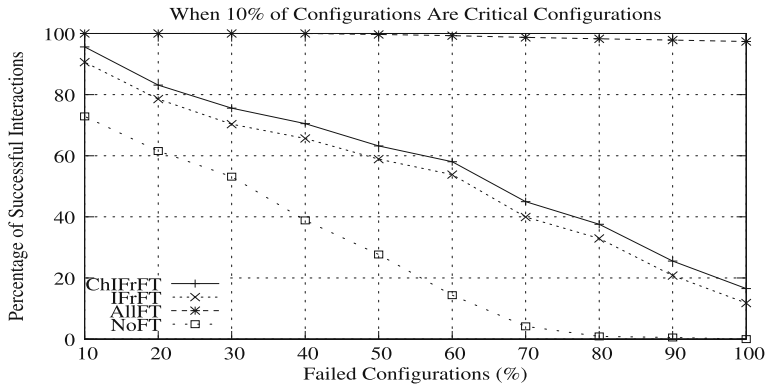

Fig. 10 Critical configurations are 10% of all configurations

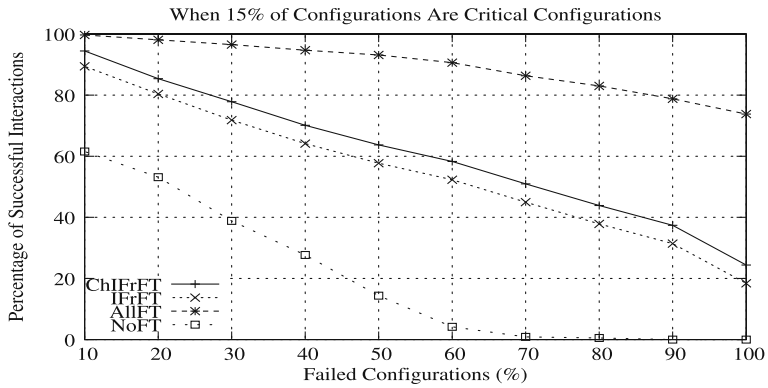

Fig. 11 Critical configurations are 15% of all configurations

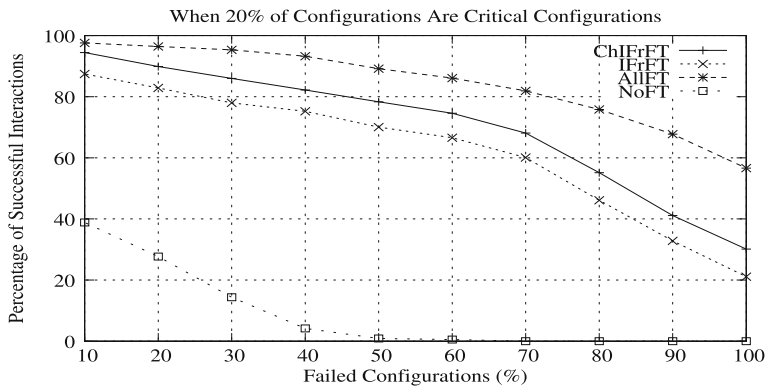

Fig. 12 Critical configurations are 20% of all configurations

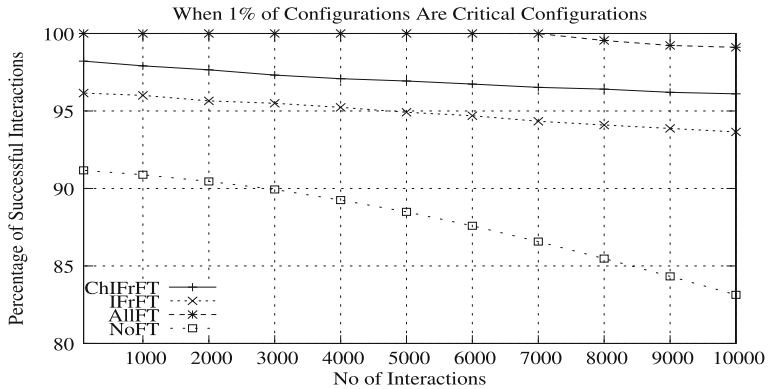

Fig. 13 Critical configurations are 1% of all configurations

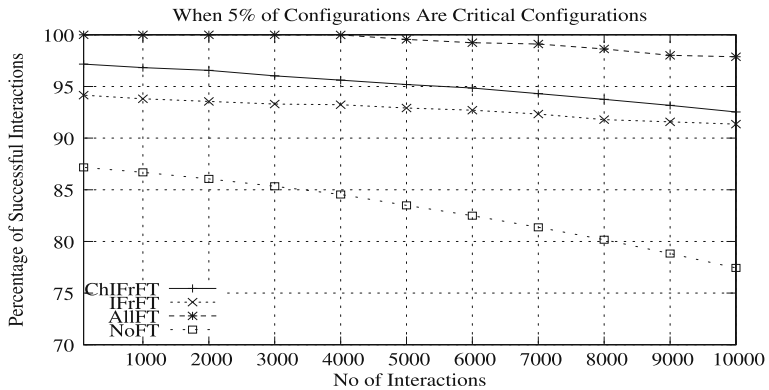

Fig. 14 Critical configurations are 5% of all configurations

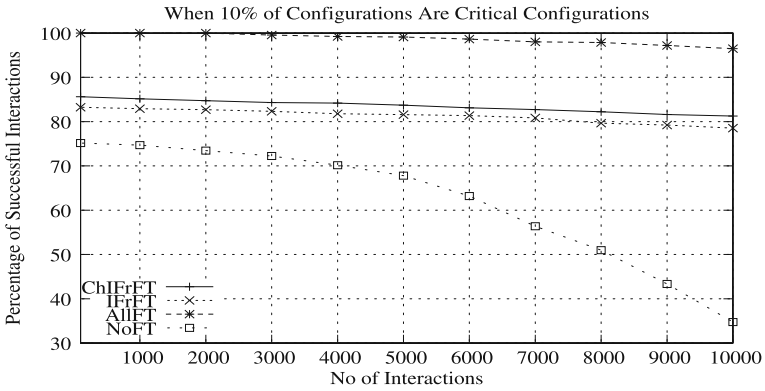

Fig. 15 Critical configurations are 10% of all configurations

- *ChIFrFT* has consistently better performance compared to *IFrFT* as the percentage of frequently used configurations increased.
- The rate at which the amount of successful interactions decrease depends on the percentage of frequently used and failed configurations.
- When frequently used configurations are 1% of all configurations of the system and the number of failed configurations increased from 10% to a hundred percent, then the number of successful interactions decrease at a faster rate as shown in Fig. 8. Reason for this is only 1% of critical configurations are supported by fault tolerant candidates. Under this scenario, when all the configurations are failed, only fault tolerant candidates which are 1% of all configurations serve requests.
- There is no much difference in percentage of successful interactions for *ChIFrFT*, *IFrFT*, and *NoFT* techniques when frequently used configurations are only 1% of all configurations as shown in Fig. 8.

Figure 9 shows the performance of a system when frequently used configurations are 5% of all configurations. As the percentage of failed configurations increased from 10 to 100 in steps of 10, the number of successful interactions will decrease. However, in this case the number of successful interactions is higher compared to the scenario where frequently used configurations are 1% of all configurations. When all the frequently used configurations are failed then fault tolerant candidates, which are equivalent to 5% of all configurations serve requests.

In this case performance of *ChIFrFT*, *IFrFT* techniques are little better than *NoFT* technique because of 5% of configurations are supported by fault tolerant candidates.

Figure 10 shows the performance of the file structure system when frequently used configurations are 10% of all configurations. In this case system performance is better compare to previous scenarios in which frequently used configurations are 1% and 5%. Furthermore, for a larger number of failed configurations *NoFT* technique suffer severely compare to *ChIFrFT* and *IFrFT* techniques.

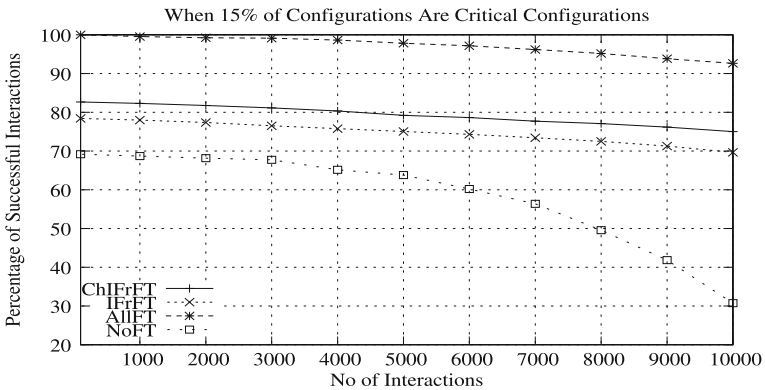

Fig. 16 Critical configurations are 15% of all configurations

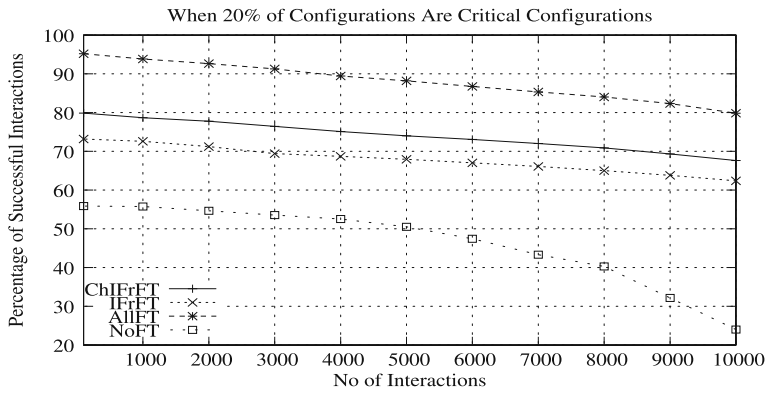

Fig. 17 Critical configurations are 20% of all configurations

Figure 11 shows the performance of the file structure system when frequently used configurations are *Fifteen* percent of all configurations. Large number of interaction requests, higher number of failed configurations lead to a large number of failed requests, even though every configuration has its replica.

As the number of interaction requests increased from 2000 to 10000 in steps of 1000, system undergo performance deterioration due to the large number of failed interaction requests. The reason is that every configuration in *AllFT* technique is supported by only one backup configuration. After primary configuration failed, its backup configuration also may fail to serve requests.

Figure 12 shows the performance of the system when frequently used configurations are 20% of all configurations. Since 20% of all configurations constitute a considerably large number of configurations that are supported by fault tolerant candidates, performance in this case is better when compared to scenarios in which critical configurations are 1%, 5%, 10%, 15% and 20%. However, 10000 interaction requests in this case have significant impact on performance of *AllFT* technique. That is, in spite of having replicas for all critical configurations they are prone to failure as the number of interaction requests increased. Also *NoFT* technique suffer from performance deterioration due to large number of interaction requests.

Figures 13, 14, 15, 16 and 17 showcase the importance of configuration interactions on performance and reliability of a software system. More the number of interaction requests, then higher the likely hood of reduced performance of a software system. Figure 13 shows the percentage of interactions that are successfully executed when frequently used configurations are 1% of all configurations. This percentage decreases as the number of interaction requests increased.

Figures 14, 15, 16, 17 shows the percentage of successful interactions when frequently used configurations are 5, 10, 15, and 20% of all configurations, respectively.

Conclusion

In this research article we have proposed fault tolerant techniques based on frequency of interactions, characteristics of configurations. *IFrFT* technique based on frequency of interactions makes use of interaction values of configurations to measure the reliability and performance of a software system. *ChIFrFT* technique is based on frequency of interactions and characteristics (critical operations such as payment transactions and control systems) makes use of structural information of configurations. Performance of *NoFT*, *AllFT*, *IFrFT* and *ChIFrFT* techniques compared for various percentages of critical configurations, number of interaction requests. While *AllFT* technique perform better than all the other techniques *ChIFrFT* approach has consistent performance compare to *IFrFT* and *NoFT*. Performance of *IFrFT* and *ChIFrFT* techniques, in terms of the number of successful interactions, has increased by 25 and 40% respectively compare to *NoFT*. Our experimental results show that either large number of interaction requests, or more number of failed critical configurations or both will result in performance deterioration of software systems in spite of having fault tolerant support for the complete software system.

Abbreviations

AllFT: All configurations enabled with fault tolerant candidates; ChIFrFT: Characteristics and interaction frequency based fault tolerance; IFrFT: Interaction frequency based fault tolerance; NoFT: None of the configurations of a software system enabled with fault tolerant candidates; RB: Recovery block; RRN: Relative record number

Acknowledgments

We thank following members of REVA University for continuous support and motivation: Our beloved Chancellor of REVA University Dr. P. Shyamaraju, Dr. V. G. Talawar, Advisor, Dr. S. Y. Kulkarni, Vice-Chancellor, Dr. M. Dhanamjaya, Registrar, Dr. N. Ramesh, Director of Placement Training and Planning, Dr. B. P. Divakar, Director of Research and Innovation, Dr. Sunilkumar S. Manvi, Director C and IT, and every member of C and IT. Also we thank Dr. N. R. Shetty, Advisor, NMIT, Dr. H. C. Nagaraj, Principal, NMIT for their constant encouragement and support. We thank G. Anitha for her comments on independent configurations and proofreading that helped us to improve the manuscript. We express our heartfelt thanks to the reviewers for their invaluable time

spent on a review process and very critical suggestions which played significant role in improving the technical content of the manuscript and also helped us in the thinking process. We thank the editor and every member of the editorial office of the Journal of Cloud Computing, Springer for their continuous support while processing research article. Last but not the least we thank Manojkumar Pandey, Arunkumar Reddy, Swetha Ramapriya, Shristi Srivastsav for their assistance in coding files structures concepts.

Funding
Not Applicable.

Availability of data and materials
Data and materials related to this research article will be uploaded as early as possible in GitHub portal.

Authors' contributions
Both authors have made intellectual contribution to the research in the field of fault tolerant computing in cloud computing. Authors have conducted experiments and responsible to writing, editing manuscript. Both authors have read this research article and approved the final manuscript.

Authors' information
Mr. Mylara Reddy C. is an Assistant Professor at School of Computing and Information Technology, REVA University, Bangalore, Karnataka, India. He received M. Tech. degree from Visvesraya Technological University in 2006. He has published few papers in international conferences and peer reviewed journals such Journal of Computer Networks, Elsevier, International Journal of Computer and Electrical Engineering. He is pursuing Ph. D. as part time research scholar under the supervision of Dr. Nalini N. at Nitte Meenakshi Institute of Technology, Bangalore, affiliated to Visvesvaraya Technological University, Karnataka, India. His research interest include cloud computing, fault tolerance, Computer networks.
Nalini N. received her Ph.D. degree in Computer Science and Engineering from Visvesvaraya Technological University in 2006. She is working as professor in the Department of Computer Science and Engineering, Nitte Meenakshi Institute of Technology, Bangalore, India. She has published several research articles in international conferences and peer reviewed journals. She has successfully supervised six students for Ph.D. degree. She is the recipient of several awards from industry, social organizations and academia. Her research interest include security in wireless communication systems, fault tolerant computing in wireless mobile systems and sensor networks, cryptography and network security, Fault tolerance framework for cloud computing applications, machine learning.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]School of Computing and Information Technology, REVA University, Rukmini Knowledge Park, Yelahanka, 560064 Bangalore, India. [2]Department of Computer Science and Engineering, Nitte Meenakshi Institute of Technology, Yelahanka, 560064 Bangalore, India.

Received: 22 March 2017 Accepted: 4 January 2018
Published online: 18 January 2018

References

1. BCzarnecki K, Eisenecker U (2000) Generative Programming: Methods, Tools, and Applications. Addison-Wesley Publishing Co.
2. Siegmund N, Rosenmuller M, Kastner C, Giarrusso PG, Apel S, Kolesnikov SS (2011) Scalable prediction of non-functional properties in software product lines. In: 2011 15th International Software Product Line Conference, Munich. pp 160–169. https://doi.org/10.1109/SPLC.2011.20
3. Batory D, Höfner P, Kim J (2011) Feature interactions, products, and composition. In: Proceedings of the 10th ACM international conference on Generative programming and component engineering. ACM, Portland. pp 13–22. https://doi.org/10.1145/2047862.2047867
4. Dart S (1990) Spectrum of Functionality in Configuration Management Systems. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-90-TR-011. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11187
5. Talbert N (1998) The cost of COTS. Computer 31: 46–52. https://doi.org/10.1109/MC.1998.683007
6. Gokhale SS, Trivedi KS (2002) Reliability prediction and sensitivity analysis based on software architecture. In: 13th International Symposium on Software Reliability Engineering. pp 64–78. https://doi.org/10.1109/ISSRE.2002.1173214
7. Wu G, Wei J, Qiao X, Li L (2007) A bayesian network based qos assessment model for web services. In: Proc. Int'l Conf. Services Computing (SCC'07). pp 498–505
8. Tsai WT, Zhou X, Chen Y, Bai X (2008) On testing and evaluating service-oriented software. IEEE Comput 41(8):40–46
9. Lyu MR (1996) Handbook of Software Reliability. McGraw-Hill, New York
10. Avizienis A (1995) The methodology and n-version programming. In: Lyu MR (ed). Software fault tolerance. Wiley, Chichester, pp 23–46
11. Rooney P (2002) Microsoft's CEO:80-20 rule applies to bugs, not just features. ChannelWeb. http://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm
12. Zhou A, Wang S, Zheng Z, Hsu C, Lyu M, Yang F (2016) On Cloud Service Reliability Enhancement with Optimal Resource Usage. IEEE Trans Cloud Comput 4(4):452–466
13. Zhang Y, Zheng Z, Lyu MR (2011) BFTCloud: a byzantine fault tolerance framework for voluntary-resource cloud computing. 2011 IEEE 4th International Conference on Cloud Computing, Washington, DC. pp 444–451. https://doi.org/10.1109/CLOUD.2011.16. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6008741&isnumber=6008659
14. Lim J, Suh T, Gil J, Yu H (2014) Scalable and Leaderless Byzantine Consensus in Cloud Computing Environments. Information Systems Frontiers, Springer 16(1):19–34
15. Ganesh A, Sandhya M, Shankar S (2014) A Study on Fault Tolerance methods in cloud computing. In: IEEE Int'l. Advanced Computing Conference (IACC):844–9. http://dx.doi.org/10.1109/IAdCC.2014.6779432
16. Vacca JR (2013) Cyber security and IT infrastructure protection. Syngress. PaperBack ISBN: 9780124166813
17. Kaur J, Kinger S (2013) Analysis of different techniques used for fault tolerance. IJCSIT: Int. J Comput Technol 4(2):737–41
18. Egwutuoha IP, Chen S, Levy D, Selic B (2012) A fault tolerance framework for high performance computing in cloud, Cluster, Cloud and Grid Computing (CCGrid). In: Proceedings of the 12th IEEE/ACM international symposium. 13-16 May. pp 709–710. https://doi.org/10.1109/CCGrid.2012.80
19. Bala A, Chana I (2012) Fault tolerance- challenges, techniques and implementation in cloud computing, ISSN (Online): 16940814. IJCSI Int J Comput Sci 9(1). www.IJCSI.org
20. Zhao W, Wenbing Z, Melliar-Smith PM, Moser LE (2010) Fault Tolerance Middleware for Cloud Computing. In: 2010 IEEE 3rd International Conference on Cloud Computing, Miami. pp 67–74. https://doi.org/10.1109/CLOUD.2010.26
21. Zhao W, Zhang H (2009) Proactive service migration for long-running Byzantine fault-tolerant systems. IET Softw 3(2):154–164
22. Nicolo P (2013) A frame work for self-healing software systems. In: IEEE 35th International Conference on Software Engineering (ICSE). pp 1397–1400. https://doi.org/10.1109/ICSE.2013.6606726
23. Saran C (2017) Cloud self-healing software be the Itdirector's way of cutting support costs? computerweekly.com. http://www.computerweekly.com/feature/Could-self-healing-software-be-the-IT-directors-way-of-cutting-support-costs
24. Nallur V, Bahsoon R, Xin Y (2009) Self-optimizing architecture for ensuring quality attributes in the cloud. In: 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, Cambridge. pp 281–284. http://doi.org/10.1109/WICSA.2009.5290820. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5290820&isnumber=5290660
25. Liu J, Wang S, Zhou A, Kumar SAP, Yang F, Buyya R (2016) Using Proactive Fault-Tolerance Approach to Enhance Cloud Service Reliability. IEEE Trans Cloud Comput PP(99):1–1. http://dx.doi.org/10.1109/TCC.2016.2567392
26. Trivedi TS (1982) Probability and Statistics with Reliability, Queuing, and Computer Science Applications. Prentice-Hall, Englewood Cliffs

27. Hecht H (1976) Fault Tolerant Software for Real-Time Applications. ACM Comput Surv 8(4):391–407
28. Li X, Qi Y, Chen P, Zhang X (2016) Optimizing backup resources in the cloud. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), San Francisco. pp 790–797. https://doi.org/10.1109/CLOUD.2016.0109. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7820346&isnumber=7820017
29. Liu J, Zhou J, Buyya R (2015) Software rejuvenation based fault tolerance scheme for cloud applications. In: 2015 IEEE 8th International Conference on Cloud Computing, New York. pp 1115–1118. https://doi.org/10.1109/CLOUD.2015.164
30. Bruneo D, Distefano S, Longo F, Puliafito A (2013) Scarpa M Workload-Based Software Rejuvenation in Cloud Systems. IEEE Trans Comput 62(6):1072–1085
31. Araujo J, Matos R, Maciel P, Matias R (2011) Software aging issues on the eucalyptus cloud computing infrastructure. In: 2011 IEEE International Conference on Systems, Man, and Cybernetics, Anchorage. pp 1411–1416. https://doi.org/10.1109/ICSMC.2011.6083867
32. Langner F, Andrzejak A (2013) Detecting software aging in a cloud computing framework by comparing development versions. In: 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent. pp 896–899
33. Clark C, Fraser K, Hand A, Hansen J, Jul E, Limpach C, Pratt I, Warfield A (2005) Live Migration of Virtual Machines. In: Proceedings of the Symposium on Networked Systems Design and Implementation. pp 273–286
34. Sapuntzakis C (2002) Optimizing the Migration of Virtual Computers. In: Proc. 5th Symp. Operating Systems Design and Implementation. Available: http://suif.stanford.edu/collective/osdi02-optimize-migrate-computer.pdf
35. Fu K, Frans Kaashoek M, Mazières D (2005) Fast and secure distributed read-only file system. In: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation, San Diego
36. Lublin U, Liguori A (2007) KVM Live Migration. KVM Forum, Tucson
37. Zhu X, Wang J, Guo H, Zhu D, Yang LT, Liu L (2016) Fault-tolerant scheduling for real-time scientific workflows with elastic resource provisioning in virtualized clouds. IEEE Trans Parallel Distrib Syst 27(12):3501–3517. https://doi.org/10.1109/TPDS.2016.2543731
38. Siegmund N, Kolesnikov S, Kastner C, Apel S, Batory D, Rosen Muller M, Saake G (2012) Predicting performance via automated feature-interaction detection. In: 2012 34th International Conference on Software Engineering (ICSE), Zurich. pp 167–177. https://doi.org/10.1109/ICSE.2012.6227196
39. Liebig J, Apel S, Lengauer C, Kastner C, Schulze M (2010) An analysis of the variability in forty preprocessor-based software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. ACM, Cape Town. pp 105–114. https://doi.org/10.1145/1806799.1806819
40. Sincero J, Schroder-Preikschat W, Spinczyk O (2010) Approaching non-functional properties of software product lines: Learning from products. In: APSEC. IEEE, pp 147–155
41. Zheng Q (2010) Improving MapReduce fault tolerance in the cloud. In: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), Atlanta. pp 1–6. https://doi.org/10.1109/IPDPSW.2010.5470865
42. Slawinska M, Slawinski J, Sunderam V (2010) Unibus: Aspects of heterogeneity and fault tolerance in cloud computing. In: Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW). pp 1–10
43. Parnas DL (1975) The influence of software structure on reliability. In: Proceedings of the international conference on Reliable software, Los Angeles. pp 358–362. https://doi.org/10.1145/800027.808458. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.645.6073&rep=rep1&type=pdf
44. Shooman ML (1976) Structural models for software reliability prediction. In: Proceedings of the 2nd international conference on Software engineering, San Fransisco, pp 268–280
45. Engelmann C, Vallee GR, Naughton T, Scott SL (2009) Proactive Fault Tolerance using Preemptive Migration. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, Weimar. pp 252–257. https://doi.org/10.1109/PDP.2009.31
46. Avizienis A (1995) The methodology of n-version programming. In: Lyu MR (ed). Software Fault Tolerance. Wiley, Chichester. pp 23–46
47. Ramos BLC (2007) Challenging malicious inputs with fault tolerance techniques. https://www.blackhat.com/presentations/bh-europe-07/Luiz_Ramos/Whitepaper/bh-eu-07-luiz_ramos-WP.pdf
48. Castro M, Liskov B (2002) Practical Byzantine Fault Tolerance and Proactive Recovery. ACM Trans Comput Syst (Assoc Comput Mach) 20(4):398–461. https://doi.org/10.1145/571637.571640. CiteSeerX: 10.1.1.127.613
49. Kim K, Welch H (1989) Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. IEEE Trans Comput 38(5):626–63
50. Nicola VF (1995) Checkpointing and the Modeling of Program Execution Time. Software Fault Tolerance. Wiley
51. Petri S, Langendörfer H (1995) Load balancing and fault tolerance in workstation clusters migrating groups of communicating processes. ACM SIGOPS Operating Systems Review 29(4):25–36. https://doi.org/10.1145/219282.219288
52. Brin S, Page L (1998) The anatomy of a large-scale hypertextual Web search engine. In: Proceedings of the seventh international conference on World Wide Web 7, Brisbane. pp 107–117. http://www.site.uottawa.ca/~diana/csi4107/Google_SearchEngine.pdf
53. Inoue K, Yokomori R, Yamamoto T, Matsushita M, Kusumoto S (2005) Ranking significance of software components based on use relations. IEEE Trans Softw Eng 31:213–225
54. Folk MJ, Riccardi G (1997) File Structures: An Object-Oriented Approach with C++. Addison-Wesley Publishing Co.