Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2014, Article ID 197423, 11 pages
http://dx.doi.org/10.1155/2014/197423

Research Article

Designing Fault Tolerance Strategy by Iterative Redundancy for Component-Based Distributed Computing Systems

Hui Wang and Yun Wang

Department of Computer Science & Engineering, Key Lab of Computer Network and Information Integration, MOE, Southeast University, Nanjing 210096, China

Correspondence should be addressed to Hui Wang; huiwang_cs@seu.edu.cn

Received 12 July 2014; Accepted 3 August 2014; Published 12 August 2014

Academic Editor: Xiaofei Zhao

Reliability is a critical issue for component-based distributed computing systems, some distributed software allows the existence of large numbers of potentially faulty components on an open network. Faults are inevitable in this large-scale, complex, distributed components setting, which may include a lot of untrustworthy parts. How to provide highly reliable component-based distributed systems is a challenging problem and a critical research. Generally, redundancy and replication are utilized to realize the goal of fault tolerance. In this paper, we propose a CFI (critical fault iterative) redundancy technique, by which the efficiency can be guaranteed to make use of resources (e.g., computation and storage) and to create fault-tolerance applications. When operating in an environment with unknown components' reliability, CFI redundancy is more efficient and adaptive than other techniques (e.g., K-Modular Redundancy and N-Version Programming). In the CFI strategy of redundancy, the function invocation relationships and invocation frequencies are employed to rank the functions' importance and identify the most vulnerable function implemented via functionally equivalent components. A tradeoff has to be made between efficiency and reliability. In this paper, a formal theoretical analysis and an experimental analysis are presented. Compared with the existing methods, the reliability of components-based distributed system can be greatly improved by tolerating a small part of significant components.

1. Introduction

With technology scaling, the occurrence of Internet-based services, such as cloud computing, volunteer computing, is sharing resources (e.g., software, hardware platform, and computation resources) to provide services on demand. At the beginning of Elastic Compute Cloud (EC2) proposed by Amazon, clouding computing which involves multiple components communication by incomplete reliable networks has become one of the hottest research areas in recent years. As a typical cloud-based application, volunteer computing uses Internet-connected computers volunteered by their owners as the source of computing power and storage. It can support applications that are significantly more data-intensive or have larger memory or storage requirements. Compared with other types of high-performance computing (e.g., grid computing), volunteer computing has a high degree of diversity. The volunteered computers vary widely in terms of software and hardware type, speed, availability, reliability, and network connectivity, as well as the resource requirements and completion time constraints of the applications [1].

The reliability of cloud computing and volunteer computing is far from perfect in reality. In traditional reliability engineering, fault-forecasting, fault-prevention, fault-removal, and fault-tolerance are used. But how to build a highly reliable and available component-based services is a challenging and urgently-demanded research problem no matter academic world or industrial community. Therefore, how to make a tradeoff between efficient use of resources and system reliability should also be taken into account. There are existing large numbers of redundant computing resources in the setting of cloud computing, especially in volunteer computing which is based on unreliable volunteer resources. A well-known technique of software fault tolerance called design diversity can be employed to tolerate faults in this setting. But when the reliability of each functionally component is low enough,

the traditional three modular redundancy may not obtain consistency results at one deployment. For instance, when the reliability of each functionally equivalent component is 0.55, the probability of three modular redundancy, that gets two or three consistency results, is $\binom{3}{2}0.55^2 \cdot 0.45 + \binom{3}{3}0.55^3 = 0.57475$. The correct result of three modular redundancy in this setting may not meet the goal of high system reliability.

We present a CFI (critical fault iterative) redundancy technique in this paper, ensuring that efficient redundancy resources can gain high system reliability. We first construct a function ranking model based on the graphic representation of the functions' invocation relationships and invocation frequencies. A function ranking algorithm is used to identify the Top-K significant functions via the invocation relationships and invocation frequencies. A new iterative redundancy technique is then proposed to enhance the system reliability, which does not require to know the component reliability (assuming the reliability of components $R_c \geq 0.5$). In this paper, the concepts of function and component are interchangeable. However, when a function executes via several functionally equivalent components, there exist some discrepancies between the reliability of the function and the reliability component. CFI, based on majority voting algorithms (such as TMR [2] and NVP [3, 4]), exploits the properties of distributed computation architectures to adapt more efficiently and to achieve the same level of system reliability at a lower cost factor. By using the function ranking algorithm, we observe that the function invoked frequently by other functions generally has a higher ranking score. On the other hand, the functions invoked by the functions with lower ranking score will get lower scores. CFI can be adapted to the dynamic environment by reexecuting the function ranking algorithm. The key property of CFI redundancy is that resources can be assigned efficiently to the most vulnerable functions to improve the system reliability. The key superiority of CFI is that it is unnecessary to know the reliability of each component. In order to show the effectiveness of the proposed method, a theoretical analysis based on probability theory and an experimental analysis based on Pajek simulation environment [5] are conducted. The CFI method can be used by the architecture designers/engineers of distributed computing or volunteering computing systems to design highly robust applications under untrusted components.

The main contributions of this paper are summarized in the following.

(i) Paper introduces a novel iterative fault tolerance strategy called CFI that does not need to know the components reliability. This expends some redundant techniques which need the components reliability and expends the scenario that iterative redundancy can be applied.

(ii) We conduct function ranking inspired by Google PageRank algorithm [6] and expand PageRank by adding invocation frequencies to better identify significance functions in complex component-based systems for redundancy. In order to make appropriate cost and reliability trade-offs.

(iii) A formal theoretical analysis based on probability theory and experiments are designed to compare the reliability of system reliability.

(iv) Extensive experiments are designed to evaluate the implicit effects of cost factor and percent of significant functions redundancy on system reliability.

The rest of this paper is organized as follows. The background and related works are introduced in Section 2. In Section 3, a system model is presented based on a ranking algorithm for searching significant functions and an iterative redundancy algorithm for fault tolerance is presented. Theoretical analysis and experiment results on present strategy are given in Section 4. Section 5 presents implicit effects on system reliability. The conclusion of the paper is shown in Section 6.

2. Background and Related Works

Many Internet services interact over unreliable networks, such as clouding computing, ecommerce, search engines, and volunteer computing. These systems utilize redundancy and replication to realize the goal of high reliability. Distributed computation architectures (DCA) systems utilize highly parallel computing resources to dynamical networks; the computing resources of DCA are built by potentially faulty and untrusted components. Widely used DCA systems such as Hadoop project [7], which uses Distributed File System (DFS) to provide high-throughput access to application data and MapReduce for parallel processing of large data sets. A form of distributed computing in which the general public volunteer processing and storage resources to scientific research project called BOINC (Berkeley Open Infrastructure for Network Computing) [8] is being used by a number of projects, including CAS@home, SETI@home, Climateprediction.net [9]. Volunteer participates provide their idle computation resources to cure diseases, study global warming, discover pulsars, and do many other types of scientific research.

Oliner and Aiken [10] propose an online, scalable method for inferring the interactions among the components of large production systems, such as supercomputers, data center clusters, and complex control systems. This work uses the idea of computing correlations and delays between component signals. Convert raw logs into meaningful anomaly signals, then use these anomaly signals to identify important relationships among components, and these relationship information is useful for system administrators to set early-warning alarms.

Automated vulnerability discovery (AVD) [11] presents a feedback-driven techniques, automatically assessing a small number of malicious participant nodes that inflict on large distributed system performance. The work focuses on the fact that the interface between correct and faulty nodes can help developers build high-assurance distributed-systems. A smart redundancy for volunteer distributed computing proposed by Brun et al. [12] demonstrates redundant strategy, which ensures efficient replication of computation and data given finite processing and storage resources. However, the

shortcoming of smart redundancy is aiming at single computing task only.

Progressive redundancy on a self-configuring optimistic programming technique aims at component-based systems proposed by Bondavalli et al. [13]. It focuses on the problem of providing tolerance to both hardware and software faults at component-based hybrid fault tolerance architecture systems. But they only consider minimizing response time and typically allocate finite resources to each task.

The motivation of this work is that intuition of failures of critical components in distributed computing system will have greater impact on system reliability; thus these critical components will have higher fault tolerance requirements. On the contrary, the other noncritical components' failure will have less impact and need less fault tolerance requirements, especially, in the circumstance that traditional three modular redundancy may not get two or three consistency results at once employment.

3. Iterative Model and Fault Tolerance Strategy

The key idea of iterative redundancy for vulnerability-driven fault tolerance strategy is made up of two steps. First of all, it identifies significant functions via invocation relationships and invocation frequencies of interconnected functions, accomplished by single component or several functionally equivalent components. Then using iterative strategy to fault tolerance unreliable components. The detailed information of these two steps is shown bellow.

3.1. Function Ranking. The purpose of function ranking is using functionally equivalent components' redundant execution to the most significant functions (or the most vulnerable functions for system reliability), in order to improve the system reliability and make the tradeoff between system reliability and efficiency. The measure, based on the invocation relationships and frequencies between interconnected functions, comes from the intuition of PageRank [6] that web pages linked by large numbers of significant pages are also important. Since the failure of these significant functions must have heavier impact on the whole system reliability than other functions, so these significant functions are more vulnerable to system reliability.

In the component-based distributed application, a weighted directed graph, called *Function Graph*, can be modeled via invocation relationships and frequencies. A node v_i in the graph represents a function accomplished by single component or several functionally equivalent components. A directed link e_{ij} from v_i to v_j represents an invocation relationship between different functions, and a nonnegative weight value $w(e_{ij})$, where $0 \le w(e_{ij}) \le 1$, represents edge weight which can be calculated by

$$w\left(e_{ij}\right) = \frac{f\left(e_{ij}\right)}{\sum_{e_{kj}\in \mathrm{IN}(v_j)} f\left(e_{kj}\right)}. \tag{1}$$

Here, $f(e_{ij})$ is invocation frequency of function pair $\langle v_i, v_j\rangle$, $w(e_{ij}) = 0$ represents that there is no invocation relationship between function v_i and v_j, and $\mathrm{IN}(v_j)$ is the set of the incoming edge of v_j. Through the definition of $w(e_{ij})$, the larger invocation ratio $w(e_{ij})$ represents that function v_j is invoked more frequently by function v_i, compared with other functions in the set of $\mathrm{IN}(v_j)$.

The weight of the function v_i is defined as the sum of the incoming edges weight $w(e_{ki})$ multiply the weight of function v_k, such that

$$w\left(v_i\right) = \sum_{e_{ki}\in \mathrm{IN}(v_i)} w\left(e_{ki}\right) \times w\left(v_k\right). \tag{2}$$

The sum weights of all function nodes in *Function Graph* is 1, such that $\sum_{v\in G} w(v) = 1$.

Based on these definitions, the procedure of component-based ranking algorithm can be computed as follows.

(i) Randomly assign an initial numerical ranking scores $w(v_i)$ to the nodes in *Function Graph*, where $0 \le w(v_i) \le 1$.

(ii) Compute the ranking score for each function v_i by the following:

$$w\left(v_i\right) = \frac{1-d}{n} + d \sum_{e_{ki}\in \mathrm{IN}(v_i)} w\left(v_k\right) w\left(e_{ki}\right), \tag{3}$$

where $n = |V|$. The parameter d is a damping factor which can be set between 0 and 1, and d is employed to adjust the significance values derived from other functions. The resulting weight values of v_i are affected by d, but the resulting ranking scores are insensitive to d. In the experiment of Section 4, when we set d from 0.7 to 0.9, the result of function ranking is stable; thus we set the parameter of d to 0.85 which is similar to [6, 14]. From (3), the weight score of function (v_i) is composed by the basic value $(1-d)/n$ and the weights score of the functions that invoked v_i. Assume that W is a vector of the functions' weight,

$$W = \begin{pmatrix} w\left(v_1\right) \\ w\left(v_2\right) \\ \vdots \\ w\left(v_n\right) \end{pmatrix}. \tag{4}$$

And D is a matrix of the invocation relationship,

$$D = \begin{pmatrix} w\left(e_{11}\right) & w\left(e_{12}\right) & \cdots & w\left(e_{1n}\right) \\ w\left(e_{21}\right) & w\left(e_{22}\right) & \cdots & w\left(e_{2n}\right) \\ \vdots & \vdots & \ddots & \vdots \\ w\left(e_{n1}\right) & w\left(e_{n2}\right) & \cdots & w\left(e_{nn}\right) \end{pmatrix}. \tag{5}$$

If v_i has no function to invoke, we set $w(v_{i1}) \cdots w(v_{in})$ to $1/n$ in general. Therefore, the simultaneous equations can be rewritten by vector form

$$\begin{pmatrix} w\left(v_1\right) \\ w\left(v_2\right) \\ \vdots \\ w\left(v_n\right) \end{pmatrix} = \frac{1-d}{n} + dD^t \begin{pmatrix} w\left(v_1\right) \\ w\left(v_2\right) \\ \vdots \\ w\left(v_n\right) \end{pmatrix}, \tag{6}$$

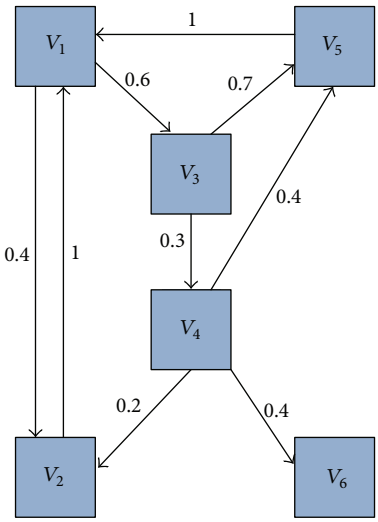

Figure 1: An example of stable invocation relationship and invocation frequency between 6 functions.

where D^t is the transposed matrix of D. If we assume that the computing process is represented by a probabilistic state transition, the function graph can be seen as a Markov chain model. Therefore the weight of each function is corresponding to the stationary state of the Markov chain.

(iii) Equation (6) can be solved by repeating the computation until all the ranking scores become stable. For the sake of simplicity, instead of repeating the computation of Markov chain's stationary state, we solve it by computing the eigenvector with eigenvalue 1 in our experiments.

Figure 1 shows a function invocation graph with computed weights. The node v_i represents the function accomplished by component, the weighted value $w(e_{ij})$ represents invocation frequency from function v_i to function v_j, and the sum of weighted values of node v_i's incoming edges is equal to 1. In this example, when setting $d = 0.85$, we will get the function significant ranking in Table 1, where V_1 invoked by V_2 and V_5 gets the highest ranking score and V_6 only invoked by V_4 gets lowest ranking score. This function ranking result is in accordance with the intuition that function invoked by significant functions is also important for system reliability.

With the approach above, Top-K most significant functions whose weight scores are highest have been identified. In the next subsection we will use redundant components' execution to enhance the reliability of the these functions to obtain higher system dependability.

3.2. Critical Fault Iterative Strategy. At the step of function ranking, Top-K significant functions have been recognized. In order to obtain high system reliability, functionally equivalent fault tolerance components can be used to meet this target. In this paper, the CFI redundant strategy is proposed to improve the system reliability efficiently. By contrast, several well-known fault tolerance techniques will be introduced,

Table 1: The function ranking and their invoked (In) and invoke (Out) degrees.

Function ID	Function Ranking	Invoked Numbers	Invoke Numbers
V_1	0.3199	2	2
V_3	0.1970	1	2
V_5	0.1797	2	1
V_2	0.1569	2	1
V_4	0.0841	1	3
V_6	0.0624	1	0

and a formal analytical analysis and a simulated empirical analysis of the system failure probability are presented.

3.2.1. Traditional Strategies

Primary Backup Replication (PBR). Primary backup replication and active replication are also well-known in the area of distributed computing. Primary backup uses serval replications to improve the system reliability. There is a replication assigned as primary. It handles on-the-fly updated of the backups to ensure limits on losses from primary replica failures, while keeping the cost of updates of the replications low. Active replication does not assigne any replica as primary replica, so it removes the centralized control of primary backup. All replicas receive system's invocation, and then reply the result. So it incurs a high cost for keeping all replicas synchronized. Active replication costs more system resources than primary backup but minimizes losses that occur when some replicas fail. Taking into account the cost of primary backup and active replication, they obtain the same failure probability, which can be calculated by $F = \prod_{i=1}^{k} f_p(i)$ where k is the number of the replicas and $f_p(i)$ is failure probability of the ith replica.

K-Modular Redundancy (KMR). K-Modular Redundancy (KMR) or N-Version Programming (NVP) are well-known fault tolerance strategies in software reliable engineering. These strategies perform $k \in \{3, 5, 7, \text{etc.}\}$, functionally equivalent and independent executions in parallel, and then take a majority voting to determine the final result. If there exists a consensus result whose votes number is bigger than $(k+1)/2$, then this consensus result is taken to be the solution. The failure of the KMR or NVP can be calculated by

$$F_{\text{KMR}}^{k} = \sum_{i=(k+1)/2}^{k} f(i), \tag{7}$$

where k is the number of functionally equivalent and independent executions and $f(i)$ is the probability that i executions are failed. Supposing the failure probability of each functionally equivalent component is $f_p(c) = 0.35$ and $k = 5$, when a component-based service distributed a job to these 5 components, then the function's failure probability is $\sum_{i=3}^{5} \binom{5}{i} 0.35^i 0.65^{5-i} \approx 0.235$. In other words, traditional

```
(1) N_a = 0;
(2) N_b = 0;
(3) result = NON;
(4) while max(N_a, N_b) < (k + 1)/2 do
(5)       distribute (k + 1)/2− max(N_a, N_b) independent jobs
(6)         to functionally equivalent components
(7)       N_a = N_a + number of a result by redundant execution
(8)       N_b = N_b + number of b result by redundant execution
(9) end
(10) result = max(N_a, N_b);
```

Algorithm 1: Progressive redundancy algorithm.

KMR or NVP fault tolerance strategies get system reliability R at cost factor k, such that

$$R_{\mathrm{KMR}}^{k} = 1 - F_{\mathrm{KMR}}^{k} = 1 - \sum_{i=(k+1)/2}^{k} \binom{k}{i} f_p(c)^i \left(1 - f_p(c)\right)^{k-i}. \tag{8}$$

Traditional redundant strategies have different advantages and disadvantages. KMR and NVP strategy must wait until all the redundant replicas have executed to determine the final result, while active replica strategy takes the first response replica as the final result. The scenarios that these redundant strategies can be employed are variant. Active replication is employed in the areas which have strict constraint of response time. Primary backup is widely used in commercial fault-tolerance systems.

3.2.2. Progressive Redundancy Strategy. Progressive redundancy strategy is a step by step calculation process, when facing component-based distributed systems whose components' reliability is high and seldom return failure results because of high reliability. In this environment, the calculation results of traditional redundancy strategy often gets consensus quickly, but it still requires to distribute jobs which will not change the task's output. Progressive redundancy strategy distributes the number of jobs to functionally equivalent components as less as possible. Taking k-majority voting for example, progressive redundancy strategy just distribute $(k + 1)/2$ jobs to component-based distributed systems. If all jobs completed by functionally equivalent components return with the same result, the consensus result will be regarded as final result, because any additional computation is irrelevant. If some functionally equivalent components (represented by R_{dis}) return with disagreeing results, the server will automatically distributes the minimum number of additional jobs, such as $(((k + 1)/2) - R_{\mathrm{dis}})$, to produce a consensus. This process is repeated until a consensus has been reached; the algorithm of progressive redundancy strategy is shown in Algorithm 1.

The reliability of progressive redundancy with k-majority voting is at most $(k-1)/2$ functionally equivalent components fail, and return disagree results:

$$R_{\mathrm{PRS}}^{k}(r) = \sum_{i=0}^{(k-1)/2} \binom{k}{i} r^{k-i}(1-r)^i, \tag{9}$$

where r represents the reliability of the functionally equivalent components and k represents cost factor.

3.2.3. Critical Fault Iterative Redundancy. The CFI redundancy will assign appropriate number of components to different functions according to the function ranking algorithm introduced above (in Section 3.1). It distributes the minimum number of functionally equivalent components to reach the system desired reliability. Since some components will fail, the results of functionally equivalent components will be variant. If all the results agree with majority components, then the task assigned to these components is completed. If some of the components fail or results disagree with majority components, then the degree of confidence of majority results is decreased. For instance, if the reliability of functionally equivalent components is 0.75, and the desired function reliability accomplished by these functionally equivalent components is 0.96. Function server distributes only one component to execute the job of this function; there is a $0.75/(0.75 + 0.25) = 0.75$ probability that the result is correct. But if the server distributes 3 functionally equivalent components to accomplish the job and all of these three components return with the same result, the degree of confidence that the consistent result is correct will be $0.75^3/(0.75^3 + 0.25^3) > 0.96$, so three is minimum number that the function to achieve confidence threshold 0.96. However, if two of three components return with agreeing results and one returns with disagreeing result, the function server at least distributes two more components return the agreeing result to achieve confidence threshold 0.96. In this scenario, how many independent components should be allocated to this function to meet the level of system reliability is determined by CFI redundancy algorithm as follows. This process can be repeated until the gap between the majority result with other results meets the requirement of the system confidence threshold.

From intuition by Bayes' Theorem, we can draw the following conclusions. If the number of the majority response results (R_{maj}) minus the other response results (R_{oth}) is constant (i.e., $R_{\mathrm{maj}} - R_{\mathrm{oth}} = C$, where C is constant), we will

```
(1)  N_a = 0;
(2)  N_b = 0;
(3)  result = NON;
(4)  while N_a − N_b < m do
(5)      redundancy execute m − |N_a − N_b| times by
         randomly choosing independent components;
(6)      N_a = N_a + number of a result by redundant execution
(7)      N_b = N_b + number of b result by redundant execution
(8)      If N_a > N_b then
(9)          result = a;
(10)     else
(11)         result = b;
(12)     end
(13) end
```

Algorithm 2: CFI redundancy algorithm.

get the same degree of confidence. For example, if a function is distributed to 10 functionally equivalent components, and 8 of them has response result A and the remaining 2 has other results, it will get the same confidence as 108 components response result A, 102 components response other results. Supposing that $m+n$ functionally equivalent components are distributed jobs to complete a given function, m components return one result with probability r, and n components return other result with probability $1-r$. $C(r,m,n)$ represents that m components reported result is correct and n components reported result is wrong (e.g., m represents result that is majority). So $C(r,m,n) = r^m(1-r)^n/(r^m(1-r)^n + r^n(1-r)^m)$. Then the proof of this Bayes' theorem, that for all j, $C(r,m,n) = C(r,m+j,n+j)$, is givin as follows:

$$\begin{aligned} C(r,m+j,n+j) &= \frac{r^{m+j}(1-r)^{n+j}}{r^{m+j}(1-r)^{n+j} + (1-r)^{m+j}r^{n+j}} \\ &= \frac{r^j}{r^j}\frac{r^m(1-r)^{n+j}}{r^m(1-r)^{n+j} + (1-r)^{m+j}r^n} \\ &= \frac{r^j}{r^j}\frac{(1-r)^j}{(1-r)^j}\frac{r^m(1-r)^n}{r^m(1-r)^n + (1-r)^m r^n} \\ &= \frac{r^m(1-r)^n}{r^m(1-r)^n + (1-r)^m r^n} = C(r,m,n). \end{aligned} \tag{10}$$

Corollary: no matter what the reliability of component ir_i is, if these components get result A $m+n$ times and get other results n times, the confidence that A is true depends only on m and is independent of n. Let X be a Bernoulli random variable that represents the number of components and let $n \in Z$; then there exists c such that, for all $m \in Z$, if out of $m+2n$ components of X, exactly $m+n$ results of components are correct, so the rest of results are wrong. Then the probability that $p(p(x) \geq 0.5)$ is constant c, because there are two possibilities: either $p(x) \geq 0.5$ or $p(x) < 0.5$. If $p(x) \geq 0.5$, the probability that exactly $m+n$ results are correct is $\binom{m+2n}{m+n} p(x)^{m+n}(1-p(x))^n$. If $p(x) < 0.5$, the probability that exactly $m+n$ results are correct is $\binom{m+2n}{n} p(x)^n(1-p(x))^{m+n}$. Then

$$\begin{aligned} &p(p(x) \geq 0.5) \\ &= \left(\left(\binom{m+2n}{m+n} p(x)^{m+n}(1-p(x))^n\right) \times \left(\binom{m+2n}{m+n} p(x)^{m+n}(1-p(x))^n + \binom{m+2n}{n} p(x)^n(1-p(x))^{m+n}\right)^{-1}\right) \\ &= \frac{p(x)^{m+n}(1-p(x))^n}{p(x)^{m+n}(1-p(x))^n + p(x)^n(1-p(x))^{m+n}} \\ &= \frac{p(x)^n}{p(x)^n}\frac{p(x)^m(1-p(x))^n}{p(x)^m(1-p(x))^n + (1-p(x))^{m+n}} \\ &= \frac{p(x)^n}{p(x)^n}\frac{(1-p(x))^n}{(1-p(x))^n}\frac{p(x)^m}{p(x)^m + (1-p(x))^m} \\ &= \frac{p(x)^m}{p(x)^m + (1-p(x))^m}, \end{aligned} \tag{11}$$

where $p(p(x) \geq 0.5)$ is only depending on m and does not depend on n. Thus, we can conclude that $p(p(x) \geq 0.5)$ is identical for all n.

Now, we have shown that the result's confidence is only depending on the different value m between the majority result with others. For instance, if m is 3 that means a function distributed to functionally equivalent components until 3 more components reported one result than the other. Then we can conduct an automatically critical fault iterative (CFI) redundant algorithm to meet the requirement of system reliability in Algorithm 2.

Using Algorithm 2, we only need to determine the system reliability requirement factor m (i.e. $m = |N_a - N_b|$); then the system reliability $R^m_{\mathrm{CFI}}(r) = r^m/(r^m + (1-r)^m)$, where r represents the reliability of the functionally

Table 2: System failure probability of different redundancy strategies.

Redundant components	Method	Component failure probability									
		1%	2%	3%	4%	5%	6%	7%	8%	9%	10%
Top 1%	CFIR	0.0833	0.1604	0.2314	0.2976	0.3588	0.4151	0.4665	0.5143	0.5581	0.5982
	RandomR	0.1379	0.2566	0.3598	0.4473	0.5237	0.5897	0.6464	0.6957	0.7374	0.7732
Top 5%	CFIR	0.0521	0.1024	0.1508	0.1973	0.2423	0.2855	0.3267	0.3662	0.4050	0.4408
	RandomR	0.1329	0.2490	0.3461	0.4361	0.5098	0.5740	0.6306	0.6796	0.7242	0.7612
Top 10%	CFIR	0.0401	0.0794	0.1179	0.1556	0.1930	0.2292	0.2648	0.2992	0.3332	0.3661
	RandomR	0.1262	0.2354	0.3320	0.4150	0.4913	0.5545	0.6149	0.6624	0.7025	0.7402
Top 0%	NoR	0.1393	0.2592	0.3624	0.4512	0.5276	0.5934	0.6501	0.6988	0.7408	0.7769
Top 100%	AllR	0.0005	0.0020	0.0045	0.0081	0.0126	0.0183	0.0250	0.0326	0.0415	0.0513

equivalent components. The algorithm first distributes m jobs to functionally equivalent components, reports minus value between the number of jobs reporting the majority results and the number reporting the other results, and then the algorithm iterates automatically distribute jobs until m more jobs have reported one result than the others.

In order to obtain the system reliability requirement factor m, $m + 2n$ jobs should be distributed to functionally equivalent components to execution, and $m + n$ functionally equivalent components return the same result, and n components return the other result. The cost factor of the iterative redundancy is shown as follows:

$$C(r) = \sum_{n=0}^{\infty} (m + 2n) \cdot \text{Prob}\,(m + n \text{ components return same result} \mid m + 2n \text{ jobs distributed to components}), \tag{12}$$

where r is the reliability of functionally equivalent components. In the case of large requirement factor m, the cost factor can be approximate compute by $C(r) \approx m/(2r - 1)$.

4. Experiment Results

In this section, we compare the improvement of system reliability based on the CFI strategy with traditional strategies and discuss the experiment results.

4.1. Experimental Framework. We use a scale-free directed function graphs generator tool called Pajek [5] to simulate component-based distributed system. A scale-free graph is a graph whose degree distribution follows a power law [15]. Large self-organizing networks, such as the Internet, the World Wide Web, and social and biological networks, often exhibit power-law degrees. Four fault tolerance approaches have been conducted to learn the performance of CFI redundancy on system reliability improvement:

(i) NoR: there is no fault tolerance strategy that is employed for the function in component based systems;

(ii) RandomR: randomly select k functions to employ fault tolerance strategy to improve the reliability of these functions;

(iii) CFIR: using the function ranking algorithm to identify the most vulnerable Top-k functions to employ iterative redundancy to improve the system reliability;

(iv) AllR: using fault tolerance strategy for all functions to obtain the system reliability.

Towards the component-based distributed system, we conduct a random trace to travel from the scale-free directed graph generated by Pajek to simulate the invocation behavior and invocation relationship. A node in the directed graph stands for the function accomplished by single component or several functionally equivalent components, an edge stands for invocation relationship, and the weight value of the edge is used to simulate the invocation probability or invocation frequency. During the execution the component-based system, initial node is randomly selected, and a random trace starting from the selected function is performed. We regard the execution as failed if the invoked function is failure; a failure probability is set to the functions provided by these functionally equivalent components. If there is a fault tolerance strategy employed for the invoked functions, the reliability of these functions will be improved. We conducted 100 travel traces for each generated scale-free directed graph. Four method, such as NoR, RandomR, CFIR, and AllR fault tolerance strategies, have been deployed for these travel traces, then averaging the simulate results.

4.2. Reliability Comparison of Distributed Computing System. When we employ different fault tolerance strategies, system will obtain different failure probabilities. The results of the experiment are showing in Table 2. In the experiment, a scale-free directed function graph with 5000 nodes is generated by Pajek. Among the experiments we simulated, AllR always gets the lowest system failure probability, while NoR always gets the highest system failure probability. The results of AllR and NoR are very intuitive, since AllR employs redundant strategies for all the functions while NoR provides no fault tolerance strategies for any function.

In the experiment, since failure probability of component is less than 1%, we just set the function requirement

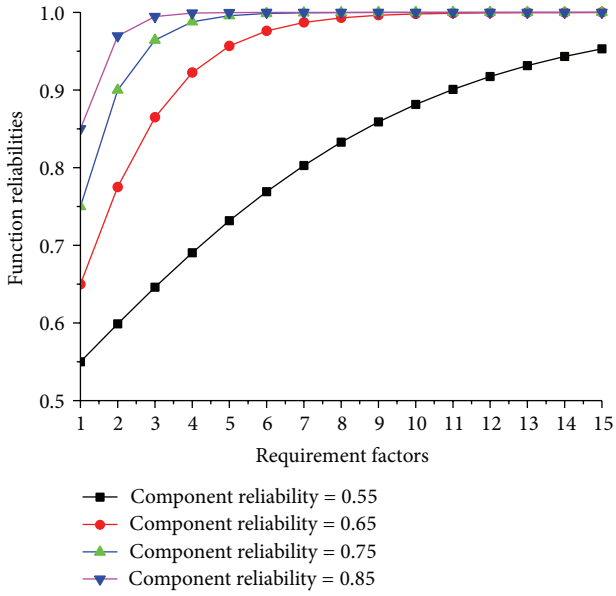

FIGURE 2: The component function reliability with requirement factor m for CFI redundancy strategy ($r = 0.55$, $r = 0.65$, $r = 0.75$, and $r = 0.85$).

factor to 3. In this setting, the function, accomplished by functionally equivalent components, will get high reliability. Compared with NoR, RandomR does not improve the system reliability obviously. This observation indicates that fault tolerance the functions that are not frequently invoked will be useless, and the failures of these nonsignificant functions will have less impact on the system reliability.

CFI redundant strategy makes a tradeoff between system reliability and cost factor. Through comparing with CFI redundant strategy, AllR obtains better system reliability in all the simulated experiments, but AllR pays a bigger price than CFI. In all our experiments, CFI fault tolerance strategy obtains better reliability than RandomR. Because significant functions identified by the step of function ranking are invoked more frequently, the failure of these significant functions will have greater impact on the component-based distributed system. So tolerating failures of these significant functions can achieve better system reliability than tolerating failures of randomly selected functions.

When the components failure probability increases from 1% to 5% and 10%, the whole system failure probabilities of four redundant strategies (e.g., NoR, RandomR, CFIR, and AllR) are increased greatly. This is because when the number of failure components increases greatly, only tolerating the failure of functions which are frequently invoked is not enough for providing a highly reliable system.

5. Implicit Effects on System Reliability

5.1. Implicit Effects of Cost Factor on Reliability. To study the impact of Cost Factor on the component-based distributed system's failure probability. Iterative redundancy method, called CFI redundancy (CFIR), proposed in this paper is compared with traditional majority voting redundancy (MajorR). The cost factor is setting from 3 to 17 with a step value of 2. The number of functions in this experiment created by Pajek is 1024. Table 3 shows that CFIR outperforms MajorR in all the cost factors no matter what redundant percent is deployed (e.g., Top 1%, Top 5%, and Top 10%). With the increase of cost factor from 3 to 17, system failure probabilities of these two redundant methods are all becoming lower.

In the corollary of system model, we have shown that it is unnecessary to know the reliability of each component to implement CFI redundant strategy (assuming the reliability of each component is bigger than 0.5). Therefore, the system architect engineer just only needs to specify how much improvement is required to enhance the system reliability. In Figure 2 we have shown that if the reliability of each component is higher than 0.75, the iterative redundant algorithm just needs to set the requirement factor to 4; then the reliability of the function accomplished by these functionally equivalent components will be higher than 0.95. The higher the component reliability, the smaller the cost factor needed to achieve the high system reliability. Therefore, if architect engineer has the knowledge of component failure probability, he may make requirement factor more effective.

In some real-time system which have strict time constraints, traditional fault tolerance strategy such as three-modular redundancy which can be deployed to three components at once, but using CFI redundant strategy, a job must first be deployed to several components, and waiting for the results before determining whether should to deploy more jobs to functionally equivalent components or not. The responding time depends on the requirement factor and component failure probability. So CFI redundancy increases the responding time for some jobs that need high reliability. In this case, more jobs can be deployed to functionally equivalent components at once to decrease the responding time.

5.2. Implicit Effects of Top-K on Reliability. In order to study the impact of the redundant percentage on system reliability, we set different redundant percents of components to compare the CFIR with MajorR. The result is showing in Table 3. The tendency of system failure probabilities when different redundant percents are deployed is shown in Figure 3.

We can conclude that when the redundant percent increases, the failure probabilities of CFIR and MajorR decrease. Under different component redundant percent settings, CFIR strategy consistently outperforms MajorR in the from Top-K = 1% to Top-K = 10%. When component failure probability is high, in order to obtain higher system reliability, larger cost factor and component redundant percent are needed.

5.3. Implicit Effects of Component Failure Probability on Reliability. We compare AllR, CFIR, RandomR, and NoR under component failure probability from 1% to 9%. The tendency of system failure probabilities when different redundant percents, which under different component failure probability settings are deployed, is shown in Figure 4. When failure

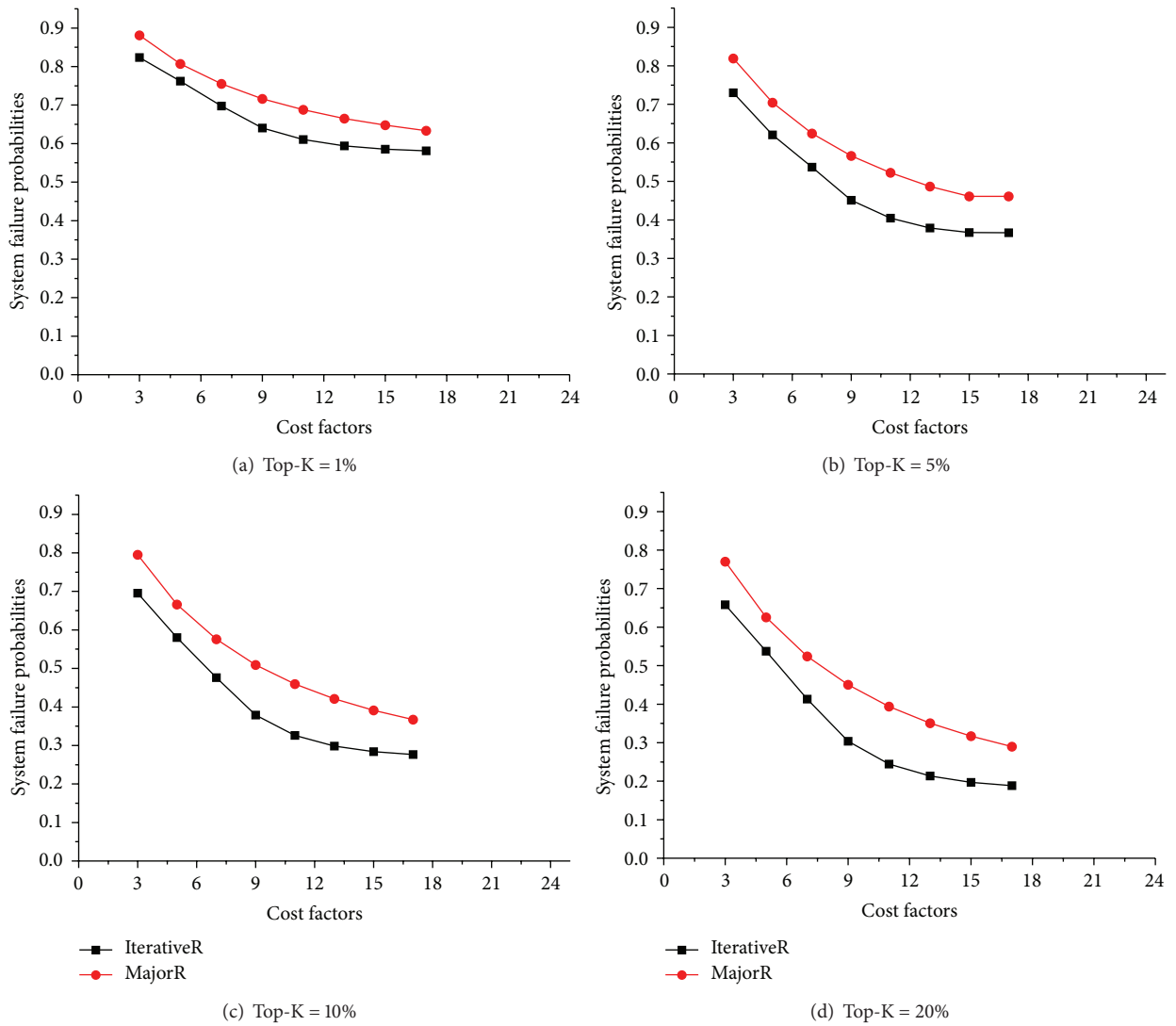

Figure 3: CFIR versus MajorR using different cost factors and redundancy percents.

Table 3: System failure probability of different cost factors with component failure probability is 0.3.

Redundant components	Method	Cost factors							
		CF = 3	CF = 5	CF = 7	CF = 9	CF = 11	CF = 13	CF = 15	CF = 17
Top 1%	CFIR	0.8237	0.7623	0.6977	0.6407	0.6107	0.5940	0.5857	0.5813
	MajorR	0.8813	0.8070	0.7550	0.7163	0.6877	0.6650	0.6477	0.6337
Top 5%	CFIR	0.7303	0.6207	0.5367	0.4510	0.4043	0.3787	0.3670	0.3667
	MajorR	0.8190	0.7043	0.6247	0.5663	0.5223	0.4870	0.4613	0.4610
Top 10%	CFIR	0.6953	0.5803	0.4763	0.3787	0.3260	0.2983	0.2840	0.2763
	MajorR	0.7953	0.6663	0.5757	0.5093	0.4593	0.4210	0.3910	0.3670
Top 20%	CFIR	0.6583	0.5373	0.4130	0.3037	0.2447	0.2133	0.1970	0.1883
	MajorR	0.7703	0.6257	0.5243	0.4503	0.3940	0.3507	0.3170	0.2900

probability of these components are increasing from 1% to 9%, the distributed system failure probability of these four methods (e.g., AllR, CFIR, RandomR, and NoR) becomes larger. CFIR outperform RandomR in all the settings and have a more effective use of redundant components.

6. Conclusion

The paper proposes a CFI redundant strategy that improves the existing techniques by using resource more efficient, especially in the environment that the failure probability of

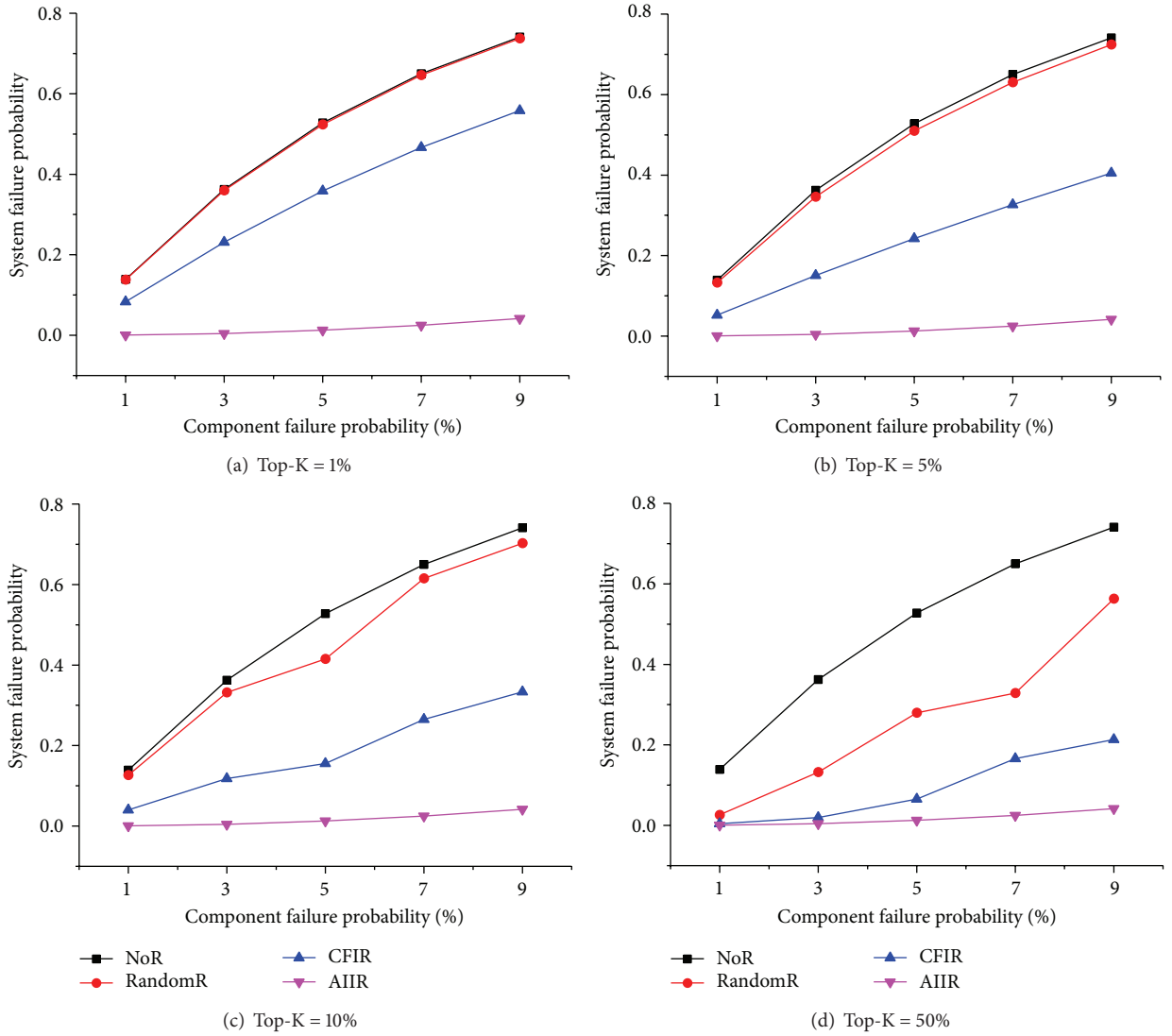

FIGURE 4: Implicit effects of component failure probability using different redundancy percents.

component is high. The CFI redundant strategy includes two steps: function ranking and iterative redundancy. In function ranking, the significant function is determined by the functions that invoke it and the weight scores of these invoke functions. At the step of iterative redundancy, different cost factors are deployed to different ranking score functions accomplished by functionally equivalent components, in order to make a tradeoff between system reliability and cost factors. In future work, when we compute the function ranking, we will consider components' failure exposure probability and failure propagation effect on ranking and considering invocation latency, concurrency, throughput, and component failure correlations when computing the weight of invocation relationship.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors thank Kun Jiang for the contributions to our modeling framework and simulation experiments, as well as Ling Zhou for the input and suggestions. This work was support by the Natural Science Foundation of China under Grant no. 60973122 and 863 Hi-Tech Program in China under Grant no. 2011AA040502.

References

[1] D. P. Anderson and G. Fedak, "The computational and storage potential of volunteer computing," in *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, vol. 1, pp. 73–80, May 2006.

[2] H. Kim, H. Lee, and K. Lee, "The design and analysis of AVTMR (all voting triple modular redundancy) and dual-duplex system," *Reliability Engineering & System Safety*, vol. 88, no. 3, pp. 291–300, 2005.

[3] L. Chen and A. Avizienis, "N-version programming: a fault tolerance approach to reliability of software operation," in *Proceedings of the 8th IEEE International Symposium on Fault-Tolerant Computing (FTCS '78)*, pp. 3–9, June 1978.

[4] A. Avizienis, "The methodology of N-version programming," *Software Fault Tolerance*, vol. 3, pp. 23–46, 1995.

[5] V. Batagelj and A. Mrvar, *Pajek: Program for Analysis and Visualization of Large Networks*, University of Ljubljana, Ljubljana, Slovenia, 2005.

[6] S. Brin, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1, pp. 107–117, 1998.

[7] D. Borthakur, "The hadoop distributed file system: architecture and design," Hadoop Project Website, 11:21, 2007.

[8] D. P. Anderson, "BOINC: a system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pp. 4–10, IEEE, November 2004.

[9] C. Christensen, T. Aina, and D. Stainforth, "The challenge of volunteer computing with lengthy climate model simulations," in *Proceedings of the 1st International Conference on e-Science and Grid Computing*, pp. 8–15, IEEE, December 2005.

[10] A. J. Oliner and A. Aiken, "Online detection of multi-component interactions in production systems," in *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN '11)*, pp. 49–60, IEEE, June 2011.

[11] R. Banabic, G. Candea, and R. Guerraoui, "Automated vulnerability discovery in distributed systems," in *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W '11)*, pp. 188–193, IEEE, June 2011.

[12] Y. Brun, G. Edwards, J. Y. Bang, and N. Medvidovic, "Smart redundancy for distributed computation," in *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS '11)*, pp. 665–676, IEEE, Minneapolis, Minn, USA, June 2011.

[13] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and J. Xu, "An adaptive approach to achieving hardware and software fault tolerance in a distributed computing environment," *Journal of Systems Architecture*, vol. 47, no. 9, pp. 763–781, 2002.

[14] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King, "Component ranking for fault-tolerant cloud applications," *IEEE Transactions on Services Computing*, vol. 5, no. 4, pp. 540–550, 2012.

[15] N. Litvak and R. Hofstad, "Degree-degree correlations in random graphs with heavy-tailed degrees," *Physical Review E*, vol. 87, Article ID 022801, 2013.