Self-Adapting Reliability in Distributed Software Systems

Yuriy Brun, *Member, IEEE*, Jae young Bang, *Member, IEEE*, George Edwards, and Nenad Medvidovic, *Senior Member, IEEE*

Abstract—Developing modern distributed software systems is difficult in part because they have little control over the environments in which they execute. For example, hardware and software resources on which these systems rely may fail or become compromised and malicious. Redundancy can help manage such failures and compromises, but when faced with dynamic, unpredictable resources and attackers, the system reliability can still fluctuate greatly. Empowering the system with self-adaptive and self-managing reliability facilities can significantly improve the quality of the software system and reduce reliance on the developer predicting all possible failure conditions. We present iterative redundancy, a novel approach to improving software system reliability by automatically injecting redundancy into the system's deployment. Iterative redundancy self-adapts in three ways: (1) by automatically detecting when the resource reliability drops, (2) by identifying unlucky parts of the computation that happen to deploy on disproportionately many compromised resources, and (3) by not relying on *a priori* estimates of resource reliability. Further, iterative redundancy is theoretically optimal in its resource use: Given a set of resources, iterative redundancy guarantees to use those resources to produce the most reliable version of that software system possible; likewise, given a desired increase in the system's reliability, iterative redundancy guarantees achieving that reliability using the least resources possible. Iterative redundancy handles even the Byzantine threat model, in which compromised resources collude to attack the system. We evaluate iterative redundancy in three ways. First, we formally prove its self-adaptation, efficiency, and optimality properties. Second, we simulate it at scale using discrete event simulation. Finally, we modify the existing, open-source, volunteer-computing BOINC software system and deploy it on the globally-distributed PlanetLab testbed network to empirically evaluate that iterative redundancy is self-adaptive and more efficient than existing techniques.

Index Terms—Redundancy, reliability, fault-tolerance, iterative redundancy, self-adaptation, optimal redundancy

1 Introduction

Developing reliable software systems is becoming more difficult as software becomes ubiquitous and is deployed on many diverse, unpredictable platforms. For example, mobiles applications need to be able to run on hundreds of different platforms. At the same time, neither cloud application developers nor users have direct access to the hardware on which the applications execute, which makes predicting possible failures harder. Finally, compromised and malicious hardware may attack the software system in unpredictable ways. This creates a software engineering challenge of building robust, reliable software systems that adapt at runtime, and on their own, to failures that are not known *a priori*.

Today, many distributed software systems, such as distributed data stores (e.g., Freenet [22]) and peer-to-peer A/V streaming applications (e.g., Skype [9]), use non-adaptive or weakly-adaptive redundancy to improve reliability. Such systems' reliability suffers when they are deployed in highly unpredictable environments, in which the identity and fraction of faulty resources are unknown, resources may join and leave at any time, and malicious nodes may unexpectedly compromise groups of agents and induce them into collusion. This resource landscape requires redundancy-based reliability techniques that are *self-adaptive* and are able to navigate the inherent unpredictability of the resources.

Iterative redundancy [14] is an efficient reliability technique applicable to a large class of distributed software systems. In this paper, we demonstrate that iterative redundancy is *self-adaptive*, and addresses system deployment in environments with unpredictable and unreliable resources. Iterative redundancy embodies a technique software engineers can use to improve the reliability of their systems, much like the traditional concept of redundancy suggests, using replicated, independent components to perform the same tasks. Unlike the traditional concept, however, iterative redundancy outperforms state-of-the-art techniques by autonomously *self-adapting* to environmental changes and not requiring *a priori* measurements of the hardware and software components' reliability. Thus, iterative redundancy simplifies the process of developing reliable software systems by ensuring reliability in an uncertain, untrusted deployment environment. Iterative redundancy has its roots in information theory and guarantees optimal use of the resources: No alternate use of the given resources can yield higher reliability. Much like the way information theory relates communication-channel entropy to the maximal information that can be transmitted over that channel, iterative redundancy relates the entropy of a *computing*

- *Y. Brun is with the School of Computer Science, University of Massachusetts, Amherst, MA 01003-9264. E-mail: brun@cs.umass.edu.*
- *J.Y. Bang, G. Edwards, and N. Medvidovic are with the Computer Science Department, University of Southern California, Los Angeles, CA 90089-0781. E-mail: {jaeyounb, gedwards, neno}@usc.edu.*

Manuscript received 22 Mar. 2014; revised 18 Dec. 2014; accepted 1 Mar. 2015. Date of publication 10 Mar. 2015; date of current version 26 Aug. 2015. Recommended for acceptance by G. P. Picco.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2412134

channel to the maximal reliability that can be achieved over that channel, even when faced with Byzantine attacks.

The goal of iterative redundancy is to simplify the task of engineering reliable, distributed software systems by providing an easy-to-implement, general approach to reliability. The key to iterative redundancy is that software systems that employ it make their decisions about resource use at the last possible moment at runtime, as opposed to at design time. This allows access to information about the reliability of various parts of the software that is not available during design, which, in turn, allows routing extra resources toward the more risky parts, and fewer resources toward the less risky parts. This ability makes iterative redundancy self-adaptive in three ways:

1) By deploying more resources when the reliability of those resources drops, the software system adapts to volatile environments and maintains high overall reliability.
2) By injecting extra redundancy into "unlucky" parts of the computation that happen to deploy on disproportionately many compromised resources, the software system ensures that these parts cannot significantly affect the overall reliability.
3) By making runtime resource allocation decisions based on the amount of agreement between the resources, the software system does not rely on often inaccurate and static *a priori* estimates of resource reliability.

We have previously shown that iterative redundancy can improve the reliability of distributed systems deployed on stable but unreliable resources [14]. Here, we focus on iterative redundancy's *self-adaptation* properties, defining iterative redundancy's two kinds of self-adaptation in Sections 4.2 and 4.3, and evaluating iterative redundancy in four ways in Section 7. Additionally, this paper formalizes iterative redundancy's optimality and formally proves the optimality claims in Section 5.

Iterative redundancy's self-adaptation allows systems that employ iterative redundancy to handle a wide range of powerful threats. Our threat model assumes that a fraction of the resources are Byzantine: they are malicious and collaborating to impede the computation. Resources may fail, or become compromised (become Byzantine) at any time. Previously unreliable resources may also suddenly become reliable. Being Byzantine, the resources may choose to pretend to be reliable for some time, and then attempt to thwart the software system whenever they decide. Further, the fraction of resources that are compromised may change at any time. Finally, the identity and fraction of the compromised resources are unknown. Iterative redundancy can adapt to all these conditions, using the available resources to optimally increase the software system reliability. That is, iterative redundancy can provide two guarantees:

1) Given a fixed increase in the number of the resources a software system may use, iterative redundancy guarantees to use those resources to produce the most reliable version of that system possible.
2) Given a fixed desired increase in the software system's reliability, iterative redundancy guarantees achieving that reliability increase with the least resources possible.

While iterative redundancy can, in principle, apply to most software systems that rely on redundancy for reliability, in this paper, we focus on a particular kind of system: *distributed computation architectures* (DCAs). DCAs solve massive problems by deploying highly parallelizable computations (i.e., sets of independent software components) on dynamic networks of potentially faulty and untrusted computing nodes. Widely known and successful DCAs include grid systems (e.g., Globus [30]), volunteer-computing systems (e.g., BOINC [11]), MapReduce systems (e.g., Hadoop [31]), and crowdsourcing applications [7], [10], [25], [28]. In the crowdsourcing domain, resources are often particularly unreliable, and reducing resource use—human time—is a critical goal. DCAs are used extensively for diverse applications, including cryptanalysis [51], web analytics [24], and scientific simulations in fields such as physics [40], astrophysics [42], bioinformatics [6], economics [36], and neuroscience [20]. Our empirical evaluation deploys one such DCA, BOINC, on the globally distributed PlanetLab testbed [47].

It is imperative for DCAs to withstand frequent failures since the entities in their networks are not subjected to any significant dependability checking and malicious entities can easily join the network and become part of the software system deployment, or compromise other participants deploying the software. Today's DCAs aim to ensure the correct execution of each task through *voting*: multiple independent worker machines perform the same computation and their results are checked for agreement. However, this technique is costly because taking a vote among n workers requires expending a factor of n resources or suffering a factor of n slowdown in performance, regardless of the circumstances, such as most of the resources being reliable or unreliable. In contrast, iterative redundancy uses those resources as efficiently as possible.

We describe iterative redundancy, formally analyze its cost and performance impact, and perform a rigorous empirical evaluation on a real-world, volunteer-computing, distributed software system. We compare iterative redundancy to two alternatives:

- *Traditional redundancy*, also called *k-modular redundancy* [38], which performs $k \in \{3, 5, 7, \ldots\}$ independent executions of the same task in parallel and then takes a vote on the correctness of the result.
- *Progressive redundancy*, which is an adaptation of a related technique from the area of self-configuring optimistic programming research [12], [13].

We demonstrate two key characteristics of resulting software systems that make iterative redundancy superior to both traditional and progressive redundancy: *self-adaptivity* and *efficiency*. Iterative redundancy is *self-adaptive* because (1) it recognizes when a computation is at a high risk of failure and injects additional redundancy to mitigate that risk, and (2) it adapts to changes in the resources' reliability. Further, iterative redundancy is more *efficient* than the two alternative methods because it produces the same level of software system reliability at a lower cost in term of employed resources (or, equivalently, higher reliability at

the same cost). In fact, iterative redundancy is optimal with respect to the cost: It is guaranteed to use the minimum amount of computation needed to achieve the desired software system reliability.

Finally, we discuss the relationship between iterative redundancy and several other types of redundancy techniques, including active replication [50], primary backup [19], checkpointing [48], and credibility-based fault tolerance [49]. In some cases, (e.g., active replication), iterative redundancy can be used in conjunction with these techniques. In other cases, (e.g., credibility-based fault tolerance), iterative redundancy can be used when these techniques cannot.

The remainder of this paper is organized as follows. Section 2 defines DCAs, the threat model, and the assumptions underlying our work. Section 3 describes the states of the practice and art in using redundancy for improving software system reliability. Section 4 describes iterative redundancy, our self-adaptive redundancy technique. Section 5 formally defines and proves iterative redundancy's optimality. Section 6 presents our empirical evaluation test beds, and Sections 7 and 8 detail empirical evaluations of self-adaptation and performance, respectively. Section 9 shows the effects of relaxing our assumptions. Section 10 places our work in the context of related research and Section 11 summarizes our contributions.

2 Scope, Threat Model, and Assumptions

This section defines our model of a DCA (the software system on which we focus our work), states the threat model we use, and enumerates the assumptions we make to aid the explanation and analysis of iterative redundancy.

2.1 System Model

In this paper, we use the following nomenclature. A *computation* is the typically large problem being solved by a DCA. A *task* is a part of the computation that can be performed independently of the others. A *job* is an instance of a task that a particular node performs. With redundancy, each task will be executed as several identical jobs on distinct nodes. In our model of a DCA, a *task server* breaks up a computation into a large number of tasks. The task server then assigns jobs to *nodes* in a node pool, ensuring that each node is chosen at random. After returning a response to a job to the task server, each node rejoins the node pool and can again be selected and assigned a new job. New volunteer nodes may join the pool while other nodes may leave.

Fig. 1 depicts the DCA software system model. The model accurately describes a number of DCAs, including the BOINC family of volunteer-computing systems [4], [11]. Section 10 discusses other distributed systems to which our techniques apply.

2.2 Threat Model

In this paper, we employ the Byzantine failure model, which is the most general and widely accepted threat model [29], [33], [38], [41] and has been applied to numerous distributed software systems [1], [3], [33]. The model includes Byzantine failures and allows for malicious nodes that collude and form cartels to try to mislead and break computations. Nodes may become or stop being Byzantine at any time. Byzantine nodes may try to report incorrect results or not report a result at all. For the purposes of this paper, we assume a node that does not report a result in a timely fashion to have failed. While at first, we will assume the fraction of nodes that are Byzantine is known, we will later show (theoretically in Section 4 and empirically in Section 7.4) that iterative redundancy does not require this information.

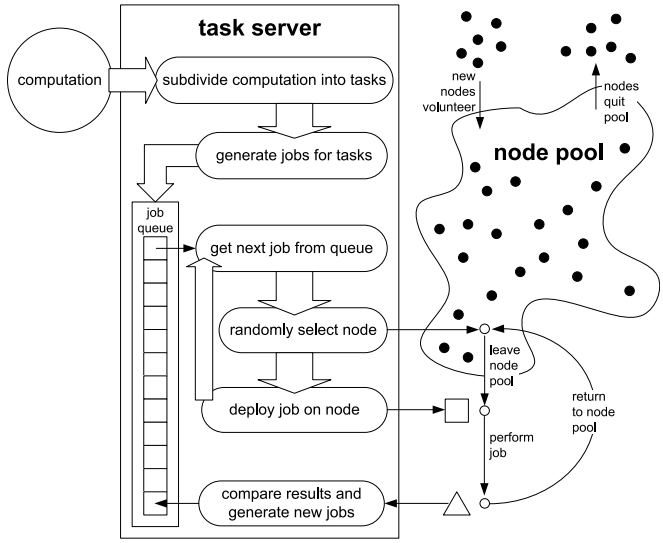


Fig. 1. A model of a DCA.

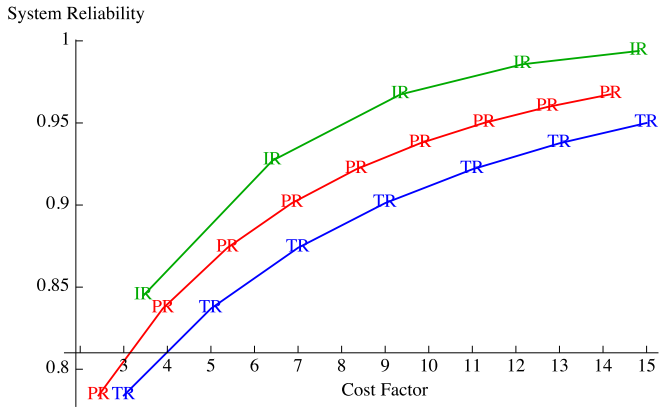
Our threat model is at least as strong as those used by redundancy techniques currently deployed in DCAs [4], [21], [24]. On the one hand, our threat model is certainly not bulletproof. For example, if failures are perfectly correlated (meaning if one node fails on a task, all nodes will fail on that task), all redundancy techniques fail to increase software system reliability. On the other hand, we make no assumptions about failures that existing implementations of DCAs do not make. In particular, we assume nodes' failures depend on the nodes, and not on the computation they perform.

Given that faults occur, our model assumes the worst possible case scenario: all faults are Byzantine faults. That is, malicious nodes may collude to return results that most hurt the reliability of the software system. For example, colluding nodes might not only return a wrong result, but the same wrong result, making it hard to identify malicious nodes. Similarly, malicious nodes are aware of other nodes that failed and how they failed, and consequently are able to return the same wrong result as those failing nodes.

In voting, the Byzantine failure model can be applied by assuming that the result of every job is one of two possible values. Although perhaps counterintuitive, this assumption creates a worst-case scenario because all failing and malicious nodes report not only a wrong result but the same wrong result, making it difficult to differentiate wrong results from correct results.

2.3 Assumptions

This section states five assumptions about the network nodes on which a DCA is deployed. These assumptions simplify the description and analysis of the three redundancy techniques, and help to define the class of distributed software systems to which the techniques apply. However,



(a) $r = 0.7$

Fig. 2. The reliability of a system approaches 1 exponentially, as a function of cost, for traditional (TR), progressive (PR), and iterative (IR) redundancy techniques (here, $r = 0.7$).

our proposed technique is not limited by these assumptions (although some of the analysis in this paper is). Section 9 will discuss relaxing these assumptions and will demonstrate that iterative redundancy still applies and, in some cases, performs even better when the assumptions do not hold.

1) Every job sent to the node pool has the same probability of failure because, even if some nodes are more reliable than others, the jobs are assigned to the nodes randomly.
2) The reliability of nodes is unknown. This assumption creates a constraint on the redundancy technique, but expands the class of software systems to which the technique applies.
3) Node failures are independent of each other. However, once nodes fail, they are allowed to collude, following the Byzantine failure model.
4) The result of every job is one of two possible values (e.g., "yes" or "no"), and the result cannot be easily verified (for example, decision NP-complete problems [52]). This assumption is derived from the Byzantine failure model, as described above.
5) The reliability of the client that receives the final result of the computation is excluded from the system's reliability.

3 Existing Redundancy-Based Reliability Techniques

This section defines two redundancy techniques: Traditional redundancy is the state of the practice for using redundancy for improving the reliability of distributed software systems. Progressive redundancy is our own adaptation of a self-configuring optimistic programming technique [12] to apply to DCA, and represents the state of the art.

To characterize the behavior of each technique, we derive formulae for two measures of their effect on software systems: the *system reliability* $\mathbb{R}(r)$ achieved by and the *cost factor* $\mathbb{C}(r)$ of applying the redundancy technique. The system reliability is the probability that a task is completed properly. The cost factor is a ratio of the number of resources needed when using the technique to the number of resources needed without it. Both of these measures are functions of the average reliability $r \in [0, 1]$ of the node pool, the fraction of time a job returns the correct response. While these formulae are at times complex, Fig. 2 provides a graphical depiction of the costs and reliabilities for $r = 0.7$. The benefits of the redundancy techniques depend on the relative improvement in reliability, rather than the absolute node reliability. Section 6 will verify the formulae's correctness experimentally.

In different domains that employ DCAs, the reliability of the nodes varies vastly. For example, in RFID networks, nodes are often 60–70 percent reliable [34]. In crowdsourcing, node reliability can vary from 70 [10] to 95 percent [8]. Internet hosts are 88 percent available [44]. Our own experimental evaluation described in Section 8.1 found that computation on distributed PlanetLab [47] nodes is 94–97 percent reliable. The redundancy techniques described here and in Section 4 are applicable to all these domains, and their benefits are similar, regardless of the underlying node reliabilities. The costs of these redundancy techniques, as we demonstrate next, are related to the relative reliability improvement, as opposed to the underlying reliability of the nodes. For example, the cost of improving from 70 to 90 percent in one domain is similar to the cost of improving from 97 to 99.9 percent in another domain. While we use, as examples, nodes with 70 percent reliability, the results generalize to more (and less) reliable nodes. Fig. 4 will show how the redundancy techniques similarly affect the relative reliability improvement under different node reliability conditions.

3.1 State of the Practice: Traditional Redundancy

The *k-vote traditional redundancy* technique (sometimes called *k-modular redundancy* [38]) performs $k \in \{3, 5, 7, \ldots\}$ independent executions of the same task in parallel, and then takes a vote on the correctness of the result. If at least some minimum number of executions agree on a result, a *consensus* exists, and that result is taken to be the solution. To simplify the subsequent discussion, we use $\frac{k+1}{2}$ (i.e., a majority) as the minimum number of matching results required for a consensus. Modern implementations of DCAs, including BOINC [4], [11] and Hadoop [31], rely on traditional redundancy.

Example. Suppose each node's reliability is $r = 0.7$ and $k = 1$ (i.e., there is no redundancy). Then the system distributes just a single job for each task and has the system reliability of 0.7. Using, instead, $k = 19$ results in a system reliability of $1-$ the chance that at least 10 of the jobs fail: $1 - \sum_{i=10}^{19} \binom{19}{i} 0.3^i 0.7^{19-i} = 0.97$, but the cost for this procedure is using 19 times as many resources.

Analysis. Recall the two measures of a redundancy technique: *system reliability* and *cost factor*. For k-vote traditional redundancy, we refer to the system reliability as $\mathbb{R}^k_{TR}(r)$ and the cost factor as $\mathbb{C}^k_{TR}(r)$. Traditional k-vote redundancy repeats every task k times, independently of r. Thus,

$$\mathbb{C}^k_{TR}(r) = k. \tag{1}$$

The reliability of k-vote traditional redundancy is the probability that at least a consensus of jobs $\left(\frac{k+1}{2}\right)$ does not fail: The sum of the probabilities that only 0, $1, \ldots,$ and $\frac{k-1}{2}$ jobs fail. Thus,

$$\mathbb{R}_{TR}^{k}(r) = \sum_{i=0}^{\frac{k-1}{2}} \binom{k}{i} r^{k-i}(1-r)^{i}. \quad (2)$$

Fig. 2 graphs the system reliability vs. the cost factor of redundancy techniques for a node pools with node reliability $r = 0.7$ The reliability of a system employing traditional redundancy (labeled "TR") approaches 1 exponentially in the cost factor.

3.2 State of the Art: Progressive Redundancy

As part of our research into redundancy techniques, we discovered a self-configuring optimistic programming technique [12] that can be redesigned to apply to DCAs. We have leveraged this scheme to develop *progressive redundancy*. To our knowledge, progressive redundancy is not used today in any deployed distributed software systems, although a related technique has been used in service-based computing [53]. We include progressive redundancy in our comparison as the state of the art because the technique on which it is based has been shown to be an improvement over traditional redundancy in other domains.

The key to progressive redundancy is the observation that traditional redundancy sometimes reaches a consensus quickly but still continues to distribute jobs that do not affect the task's outcome. Progressive redundancy minimizes the number of jobs needed to produce a consensus: The k-vote progressive redundancy task server distributes only $\frac{k+1}{2}$ jobs. If all jobs return the same result, there will be a consensus and the results produced by any subsequent jobs of the same task become irrelevant. If some nodes agree, but not enough to produce a consensus, the task server automatically distributes the minimum number of additional copies of the job necessary to produce a consensus, assuming that all these additional executions were to produce the same result. The task server repeats this process until a consensus is reached.

Example: As before, suppose $k = 19$ and $r = 0.7$. Using progressive redundancy, the system reliability is the probability that fewer than 10 (fewer than half) of the jobs fail, or 0.97, which is the same as traditional redundancy. As we will show in Equation (3), the cost of this procedure is using 14.2 times as many resources as a system without redundancy. This number is 1.3 times smaller than the cost of traditional redundancy: while sometimes a task is distributed to as many as 19 nodes, many tasks reach the consensus earlier.

Analysis. For k-vote progressive redundancy, we use $\mathbb{R}_{PR}^{k}(r)$ and $\mathbb{C}_{PR}^{k}(r)$ to denote the system reliability and the cost factor, respectively. The cost factor of progressive redundancy is at least the consensus (since at least that many jobs must be distributed), plus the sum, for every integer i larger than the consensus up to k, of the probability that i jobs have not produced a consensus. Thus,

$$\mathbb{C}_{PR}^{k}(r) = \frac{k+1}{2} + \sum_{i=\frac{k+3}{2}}^{k} \sum_{j=i-\frac{k+1}{2}}^{\frac{k-1}{2}} \binom{i-1}{j} r^{i-1-j}(1-r)^{j}. \quad (3)$$

The reliability of a system with k-vote progressive redundancy is the probability that at least a consensus of jobs $\left(\frac{k+1}{2}\right)$ do not fail, exactly the same as with traditional redundancy:

$$\mathbb{R}_{PR}^{k}(r) = \sum_{i=0}^{\frac{k-1}{2}} \binom{k}{i} r^{k-i}(1-r)^{i}. \quad (4)$$

Fig. 2 shows that for a given cost factor, progressive redundancy (labeled "PR") always achieves a higher system reliability than traditional redundancy.

4 Self-Adaptive, Iterative Redundancy

DCAs typically execute jobs asynchronously and have (1) access to runtime information about system reliability and (2) the ability to alter task deployment based on that information. We have used this observation to develop *iterative redundancy*.

4.1 Iterative Redundancy Definition and Analysis

Iterative redundancy distributes the minimum number of jobs required to achieve a desired confidence level in the result, assuming that all the jobs' results agree. Then, if all jobs agree, the task is completed. However, if some results disagree, the confidence level associated with the majority result is diminished. The algorithm then reevaluates the situation and distributes the minimum number of additional jobs that would achieve the desired level of confidence, given the prior results. This process iterates until the majority agreeing results sufficiently outnumber the minority disagreeing results to reach the confidence threshold.

Example. Suppose $r = 0.7$ and the desired system reliability is $\mathbb{R} = 0.97$. Iterative redundancy uses $\mathbb{R}$ as the confidence threshold and calculates how many jobs' results must unanimously agree to be $\mathbb{R}$ confident in result's correctness. For example, if the task server distributes only one job, there is a $\frac{0.7}{0.7+0.3} = 0.7$ chance that the result is correct, but if the task server distributes five jobs and they all return the same result, there is a $\frac{0.7^5}{0.7^5+0.3^5} > 0.97$ chance that the result is correct. Five is the minimum number of jobs that can achieve the confidence threshold in this example, so the task server distributes five jobs. If all five jobs return the same result, the task is finished. However, if some jobs return a result that disagrees with the majority, the task server determines the minimum number of additional jobs that must be distributed to achieve the confidence threshold and produce the desired system reliability. For example, if four jobs return agreeing results and one returns a disagreeing result, the task server determines that at least two more jobs must return the majority result (with no additional jobs returning the minority result) to achieve $\mathbb{R}$. The task server then automatically distributes two more jobs. As we will show in Equation (5), the cost of iterative redundancy, for this particular example, is the use of 9.4 times as many resources as a system without redundancy. Note that this cost is 1.5 times less than the cost of progressive redundancy and 2.0 times less than the cost of traditional redundancy. However, iterative redundancy also increases the latency of the computation, as described in Section 8.2.

```
COMPUTE(Task task, int d)
1   a ← 0
2   b ← 0
3   while a − b < d
4     deploy d − (a − b) task jobs on
5       independent, randomly chosen nodes
6     a ← a + number of a results returned
7     b ← b + number of b results returned
8     if a < b
9       a ↔ b
10  return result a
```

Fig. 3. The iterative redundancy algorithm.

Intuitively, progressive redundancy is guaranteed to distribute the fewest jobs to achieve a consensus. In contrast, iterative redundancy is guaranteed to distribute the fewest jobs needed to achieve a *desired system reliability*. Thus far, we have avoided specifying how the technique determines this minimum number of jobs. The basic intuition described above leads to an algorithm that requires (1) numerous relatively complex probability computations and (2) node reliability as an input parameter. Requiring the availability of node reliabilities violates one of the assumptions we stated in Section 2.3. We made this assumption because, for many systems, it is not practical to obtain this information. For example, in a volunteer-computing system, the system has no information about new volunteers. If the system collects information about the reliability of nodes over time, malicious nodes that have developed a bad reputation can change their identity. For iterative redundancy, we have devised an algorithm that does not require knowledge of node reliability and can thus be applied to a wider class of systems than credibility-based fault tolerance and blacklisting [49].

Algorithm. Iterative redundancy takes an argument d, which corresponds to a measure of how much reliability improvement is desired. This situation is parallel to the progressive and traditional redundancy techniques, in which the user specified a parameter k. (If the node reliability r and the desired system reliability $\mathbb{R}$ are known, d is the minimum number of jobs that have to agree unanimously to achieve the desired confidence level. The system reliability is the certainty that all d jobs succeeded, as opposed to all d jobs failed, thus d must be minimal, such that $\frac{r^d}{(1-r)^d+r^d} \geq \mathbb{R}$.) Iterative redundancy deploys d jobs, waits for the responses, computes the distribution of those responses, and then deploys the minimum number of jobs such that, if all the jobs respond with the most-frequent answer, there will be d more such answers than other answers. Fig. 3 specifies the iterative redundancy algorithm in pseudocode. We have previously, formally proven that this algorithm results in the optimal number of job distributions, and that the answer confidence is sufficiently high, whenever the number of most-frequent answers is at least d larger than the other answers [14]. We omit this proof here.

Analysis. For iterative redundancy with d as defined above, we use $\mathbb{R}^d_{IR}(r)$ and $\mathbb{C}^d_{IR}(r)$ to denote the system reliability and the cost factor, respectively. The cost factor of iterative redundancy is the sum, for every b, of the probability that the system distributes $(d+2b)$ jobs and receives $d+b$ of one result and b of the other, weighted by the cost $(d+2b)$. Thus,

$$\mathbb{C}^d_{IR}(r) = \sum_{b=0}^{\infty}(d+2b)P\left[\begin{matrix}d+2b \text{ jobs produce}\\ d+b \text{ identical results}\end{matrix}\right] \approx \frac{d}{2r-1}. \quad (5)$$

Finally, the reliability of a system with iterative redundancy is the probability that d more jobs return the right result than the wrong result. Thus,

$$\mathbb{R}^d_{IR}(r) = \frac{r^d}{r^d+(1-r)^d}. \quad (6)$$

Note that $\mathbb{R}^d_{IR}(r)$ depends only on the difference d between the majority and minority counts of the responses. This result, proven in [14], follows from Bayes's Theorem.

Fig. 2 shows that for a given cost factor, iterative redundancy (labeled "IR") always achieves a higher system reliability than both traditional and progressive redundancy.

Fig. 2 compares the three redundancy techniques for a single node reliability of $r = 0.7$, which is realistic in some domains [10], [34]. However, iterative redundancy's benefits extend to other domains with different underlying node reliabilities. Fig. 4 demonstrates these benefits for four domains with node reliabilities of $r = 0.7$, 0.8, 0.9, and 0.97 (the latter is the highest empirically measured reliability we observed on the PlanetLab [47] nodes, as described in Section 8.1). While the probability scale shifts in these log plots, the relative benefits of iterative redundancy translate across domains: Iterative redundancy saves as many resources when a 99.9 percent-reliable computation needs to be composed of 97 percent-reliable nodes, as when a 90 percent-reliable computation needs to be composed of 70 percent-reliable nodes. Of course, if the underlying resources are as reliable as the desired reliability of the computation, there is no need for redundancy techniques of any kind. If, however, the computation needs to be more reliable than the resources, redundancy techniques can help, and

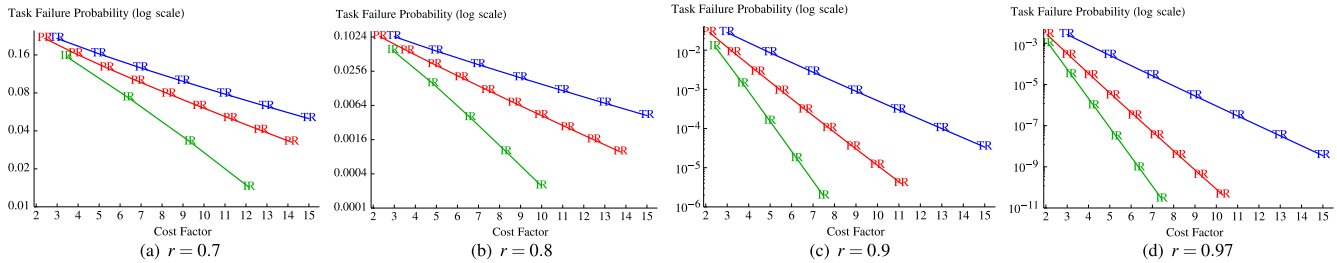


Fig. 4. Iterative redundancy (IR) produces lower probability of task failure for the same cost than progressive (PR) and iterative (TR) redundancy. This reduction holds for domains of varying underlying node reliability; Fig. 10c summarizes the benefits of iterative redundancy over a wide range of node reliabilities.

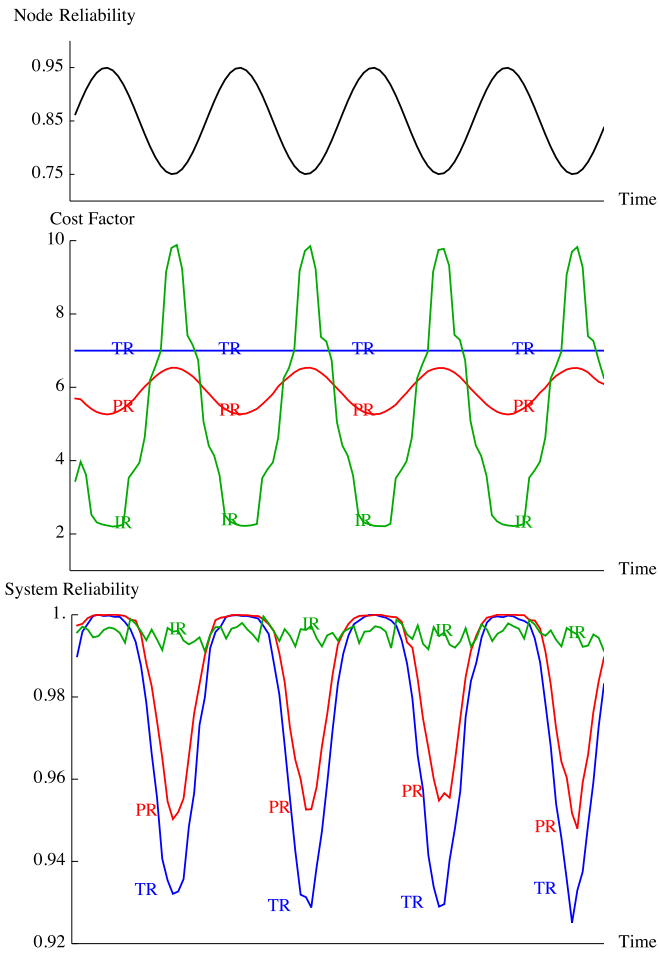


Fig. 5. Traditional, progressive, and iterative redundancy techniques react differently to the changing node reliability r. As r changes (top), traditional redundancy keeps a constant cost while progressive and iterative redundancy adapt by executing more jobs (middle). When r drops, however, traditional redundancy allows system reliability to drop significantly, progressive redundancy allows system reliability to drop slightly, while iterative redundancy keeps system reliability fairly constant (bottom).

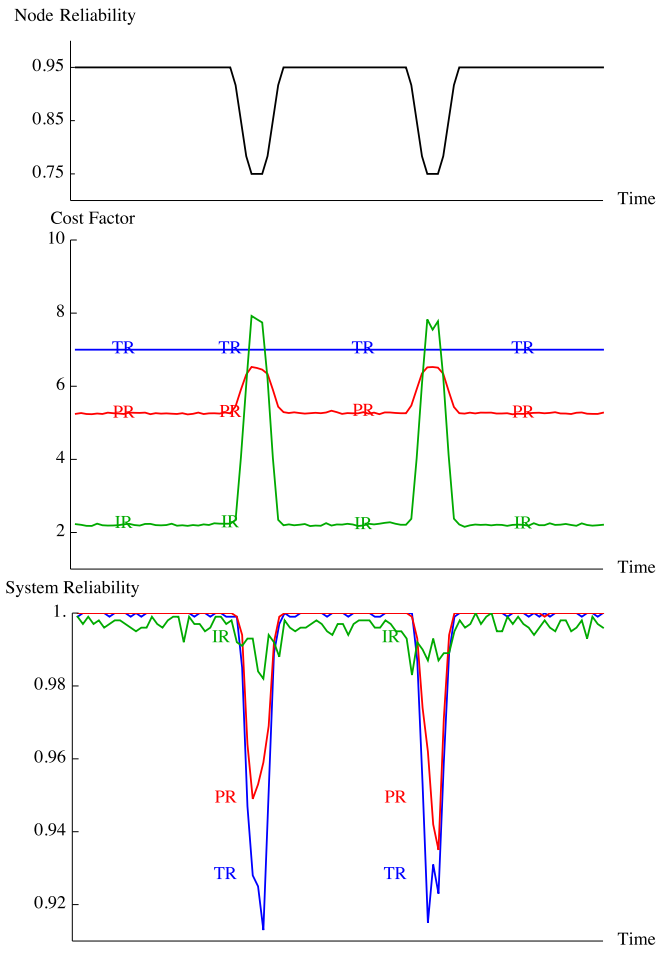


Fig. 6. The self-adaptive behavior of iterative redundancy is even more evident when the node reliability (top) changes rapidly. When the node reliability drops, iterative redundancy cost spikes up (middle), but keeps the system reliability (bottom) high.

iterative redundancy's benefits over other redundancy techniques increase as the need for reliability increases.

4.2 Self-Adaptation to Changing Resource Reliability

Iterative redundancy is self-adaptive because it automatically adjusts to changes in its environment. When the reliability of the underlying nodes drops, the disagreement between the nodes automatically triggers the addition of extra resources to the computation. Similarly, when the underlying node reliability increases, the decrease in disagreement automatically withholds extra resources, ensuring efficiency. Unlike other redundancy techniques, iterative redundancy can be used to enforce a consistent system reliability, even when the reliability of the underlying resources varies.

Because DCA task servers assign jobs to nodes randomly (recall Section 2.1), even if the reliability of the nodes remains a constant throughout the execution, some tasks may be unlucky and receive disproportionally many faulty resources. Iterative redundancy automatically adjusts to these situations as well, again, with the increased disagreement triggering the deployment of extra resources for these unlucky situations, guaranteeing not only that the overall system reliability remains high on average, but also that the variance between the reliability of different tasks remains low. Iterative redundancy remains efficient by withholding extra resources from tasks that by luck are assigned to few failing resources.

Section 7 will empirically evaluate the self-adaptive properties of iterative redundancy, demonstrating what happens when resource reliability changes both gradually and rapidly, and how iterative redundancy handles lucky and unlucky situations. Figs. 5 and 6 will show that iterative redundancy's self-adaptation can maintain a near-constant system reliability even when faced with unreliable resources and resources whose reliability changes over time; traditional and progressive redundancy techniques cannot accomplish the same.

4.3 Self-Adaptation to Unknown Resource Reliability

Even if the reliability of the resources does not change during the execution, the reliability may not be known at runtime. In such situations, redundancy techniques can struggle with computing how much redundancy is needed, and may yield systems that are not reliable enough, or are more reliable than required and thus inefficient.

Software systems employing iterative redundancy use the agreement between the resources to automatically measure and handle the resource unreliability. This enables using the fewest possible resources to produce a system at the desired level of reliability without using inaccurate *a priori* estimates of resource reliability. Section 7 will empirically evaluate this claim, and Fig. 9 will demonstrate that systems employing iterative redundancy act identically when given under- and overestimates of resource reliability: They use the resources optimally to get as reliable a system as is possible.

5 Iterative Redundancy Optimality

Under the model and the assumptions from Section 2, iterative redundancy is optimal in two ways: First, given a desired increase the system reliability, iterative redundancy deploys the fewest possible jobs to achieve that increase. Second, given a set of resources that can perform a fixed number of jobs, iterative redundancy uses them to produce the highest possible overall system reliability. We now formally state and prove these two claims.

Theorem 1. *Let $\mathbb{R}$ be a desired system reliability and let r be the average node reliability. Let t be a task and $J = \langle j_1, j_2, j_3, \ldots, j_n \rangle$ be the set of n jobs iterative redundancy deploys in computing t with reliability $\mathbb{R}$. Then all possible deployments of $n' < n$ jobs would result in a system reliability $\mathbb{R}' < \mathbb{R}$.*

Proof (by contradiction) . Let us assume that there exists a deployment of $n' < n$ jobs that results in a system reliability of $\mathbb{R}$ or higher. Consider the execution of the iterative redundancy algorithm from Fig. 3, focusing on the last iteration of the while loop on lines 3–9. If the iterative redundancy algorithm were executing, the final iteration would deploy $d - (a - b)$ jobs on line 4 and bring the total number of deployed jobs to n. So, regardless of what happens on the first $n - (d - (a - b))$ jobs in all deployments, to only deploy n' jobs total, the algorithm would have to finish by deploying at most $d - (a - b) - 1$ jobs. But then, even if all the jobs returned the same result, the largest value $|a - b|$ could be is $d - 1$. But by definition, d is minimal such that $\frac{r^d}{(1-r)^d + r^d} \geq \mathbb{R}$. Therefore, since $n' < n$ implies $|a - b| < d$ and $|a - b| < d$ implies $\mathbb{R}' < \mathbb{R}$, then the deployment of $n' < n$ jobs results in a system reliability $\mathbb{R}' < \mathbb{R}$. Contradiction. □

Theorem 2. *Let $T = \langle t_1, t_2, t_3, \ldots, t_m \rangle$ be a set of m tasks that need to be computed for a computation, and let $\frac{n}{m}$ be the cost factor iterative redundancy may use for the computation (i.e., iterative redundancy may deploy a total of n jobs). And let the minimum reliability of the k tasks that iterative redundancy produces be $\mathbb{R}$. Then no other assignment of the n jobs to the k tasks can result in a larger minimum $\mathbb{R}'$.*

Proof. Let us consider the assignment of jobs to tasks that iterative redundancy produces, focusing on the task $\hat{t}$ that produces the lowest overall reliability $\hat{\mathbb{R}}$. By definition, $\hat{\mathbb{R}} = \mathbb{R}$. All possible reassignments of jobs to tasks can be classified into three categories, ones that do not affect $\hat{t}$'s reliability, ones that reduce $\hat{t}$'s reliability, and ones that increase $\hat{t}$'s reliability.

1) All reassignments that do not affect $\hat{t}$ cannot increase the minimum reliability $\mathbb{R}'$ because increasing other tasks' reliability will keep $\mathbb{R}' = \hat{\mathbb{R}}$, and reducing other tasks' reliability can only reduce or keep the same the overall minimum.
2) All reassignments that reduce the reliability of $\hat{t}$ reduce $\mathbb{R}'$ because at least one task, $\hat{t}$, will have a lower reliability than $\mathbb{R}$.
3) Finally, all reassignments that increase the reliability of $\hat{t}$, by Theorem 1, must increase the number of jobs assigned to $\hat{t}$, and therefore reduce the number of jobs assigned to the other tasks, and therefore at least one task must have at least one less job assigned to it. Let us consider that task $\check{t}$. By Theorem 1, iterative redundancy used the minimum number of jobs possible to achieve a reliability of $\check{t}$ that is at least $\mathbb{R}$, and reducing the number of jobs is guaranteed to produce a reliability $\check{\mathbb{R}}' < \mathbb{R}$, thus reducing the overall minimum $\mathbb{R}'$ below $\mathbb{R}$. □

The implications of Theorems 1 and 2 are that as long as every deployed job has the same probability of failing (or rather, as long as the system employing iterative redundancy does not have sufficient information to distinguish deployed jobs' reliability differences), iterative redundancy is optimal. In DCAs, and under the Byzantine threat model, this assumption holds. However, as Section 9.1 will discuss, weaker threat models and known dependences between job deployments can lead to more efficient techniques. For example, if the knowledge that a node recently returned a job result that disagreed with other nodes' results can be used to predict that that node's next job's result will likely also disagree, this information can be used to further reduce the resource use. However, Byzantine nodes that can pretend to be reliable, only to fail at the most inopportune moment, can thwart this attempted efficiency improvement.

6 Evaluation Platforms

We evaluated traditional, progressive, and iterative redundancy theoretically, based on Equations (1) through (6), and empirically, using a discrete event simulation of a DCA, and a deployment of the BOINC volunteer-computing software system [4], [11] on the distributed PlanetLab platform [47]. Further, we used off-the-shelf distributed systems to evaluate the redundancy techniques: XDEVS [27] and BOINC [11].

6.1 XDEVS Simulation Environment

The XDEVS simulation framework [27] is a highly extensible discrete event simulator specialized for simulating software systems. Unlike other discrete event simulators, XDEVS provides a software-oriented programming model by supporting abstractions commonly used in software design models (e.g., components, interfaces, and resources) as first-class modeling entities. We modeled the task server as an XDEVS component and the node pool as an XDEVS resource. The jobs distributed to nodes in our XDEVS simulations do not solve any specific problem; rather, they perform simulated work for a simulated period of time. The XDEVS simulation engine, which is designed to enforce

constraints on system behavior, ensures that our system model described in Section 2 is accurately represented.

Using XDEVS allowed us to rapidly implement each redundancy technique, flexibly experiment with system parameters, such as the job reliability and amount of redundancy employed, and observe dynamic behavior not exposed by formal static analysis. To allow for comparison, all the data in Sections 7 and 8 were generated from XDEVS simulation runs with (1) at least 1,000,000 tasks and 10,000 nodes, (2) job completion times that varied stochastically between 0.5 and 1.5 time units, according to a uniform distribution, and (3) an average node reliability of 0.7 (except where explicitly noted otherwise).

Each simulation run recorded the simulated time units required to complete the computation, the total number of jobs generated, the average number of jobs per task generated, the maximum number of jobs generated for any single task, the number of tasks that achieved a correct result, the average response time per task, and the maximum response time for any task.

6.2 BOINC Deployment

Our second empirical evaluation utilized the BOINC volunteer-computing system [4], [11]. BOINC is a popular DCA currently deployed on over a million machines. Examples of BOINC applications include SETI@home, Folding@home, Malariacontrol.net, and Climateprediction.net. The BOINC server software [11] allows distribution of a custom problem to volunteering computers. To compare the three redundancy techniques, we (1) developed a custom task server that decomposes 3-SAT [52] problems into individual tasks that test if particular Boolean assignments satisfy a Boolean formula, and (2) modified the job-assignment and result-validation procedures to employ iterative and progressive redundancy. We have made all our source code and BOINC modifications publicly available and open-source at http://softarch.usc.edu/~ronia/sr/.

One aspect of the iterative and progressive redundancy algorithms that make them attractive for use in software systems is that they are simple to understand and to implement. For example, extending BOINC to enable it to use different redundancy schemes required modifying only 153 lines of code. Adding the iterative redundancy mechanism required modifying an additional 39 lines of code, while adding the progressive redundancy mechanism required modifying an additional 41 lines of code. These modifications are all available at http://softarch.usc.edu/~ronia/sr/. This anecdotal evidence suggests that legacy software systems can be extended with relative ease to use other redundancy schemes, and iterative redundancy in particular. We believe implementing iterative redundancy into newly developed systems is similarly easy.

We deployed BOINC on a 200-node subset of PlanetLab [47]. The PlanetLab testbed consists of $\sim 1{,}000$ machines of varying speed and resources, distributed at ~ 500 locations around the world. Despite some well-known issues with PlanetLab, such as unresponsive nodes and heavy utilization [26], it served us as a reasonable testbed.

In deploying BOINC on PlanetLab, we uncovered that BOINC employs two levels of redundancy: every task is deployed as k jobs, but also, every job is deployed several times in case some nodes executing the job crash or fail to return a result. BOINC is thus forced to waste considerable resources in order to avoid failures. Iterative redundancy can handle nodes returning incorrect results as well as nodes not returning results (with proper time-out mechanisms), reducing the use of resources even further.

To allow for comparison, all the data in Sections 7 and 8 were generated from BOINC executions on 200 nodes that solved 22-variable 3-SAT problems. Each problem was decomposed into 140 tasks. Three types of failures were present in the BOINC system:

1) seeded failures that caused the wrong result to be returned 30 percent of the time,
2) PlanetLab nodes becoming unresponsive, and
3) all other unanticipated failures that PlanetLab nodes might experience.

Each execution recorded the time to complete the computation, the total number of jobs generated, the average number of jobs per task generated, the maximum number of jobs generated for any single task, and the number of tasks that achieved a correct result.

7 Self-Adaptation Evaluation

Two of the ways in which iterative redundancy adapts to a changing environment are: (1) automatically increasing the number of jobs per task when node reliability drops and decreasing the number of jobs per task when node reliability rises, and (2) injecting extra redundancy into "unlucky" situations with disproportionately many failures.

While progressive redundancy also exhibits this property to a limited extent, progressive redundancy bounds both the minimum and maximum number of jobs per task. For example, when node reliability is close to 0.5, progressive redundancy tends to use about k jobs for each task, while when node reliability approaches 1, it uses about $\frac{k+1}{2}$ jobs per task. Iterative redundancy does not bound the maximum number of jobs deployed per task; however, as the number of tasks increases, the cost factor—the average number of jobs deployed per task—approaches $\mathbb{C}_{IR}^{d}(r)$.

To illustrate how iterative redundancy automatically adjusts the number of jobs per task to changes in node reliability, we conducted XDEVS-based experiments, varying the reliability of the nodes over time. Sections 7.1 and 7.2 describe experiments in which the node reliability varies gradually and rapidly, respectively.

7.1 Gradually Varying Reliability

Fig. 5 shows how a gradually changing node reliability affects the cost factor and the system reliability, using each of the three redundancy techniques. Node reliability (top), varies between 0.75 and 0.95 gradually, and is the same for all executions. However, the average jobs per task (middle), and the percentage of tasks that return a correct result (bottom), are quite different for each technique. Traditional redundancy keeps a constant number of jobs per task, but as the reliability r of the underlying nodes drops, the percentage of tasks yielding a correct result drops significantly. Progressive redundancy allows the jobs per task to vary within a predefined range to adjust to changes in

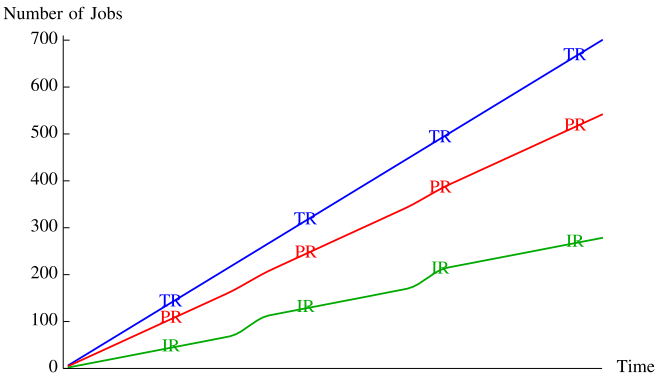


Fig. 7. Despite injecting extra redundancy when the node reliability drops, iterative redundancy saves significant effort by reducing the effort when extra redundancy is not needed. For the reliability scenario from Fig. 6 (the time scale follows that figure), iterative redundancy requires 2.5 and 1.3 times fewer jobs than traditional and progressive redundancy, respectively.

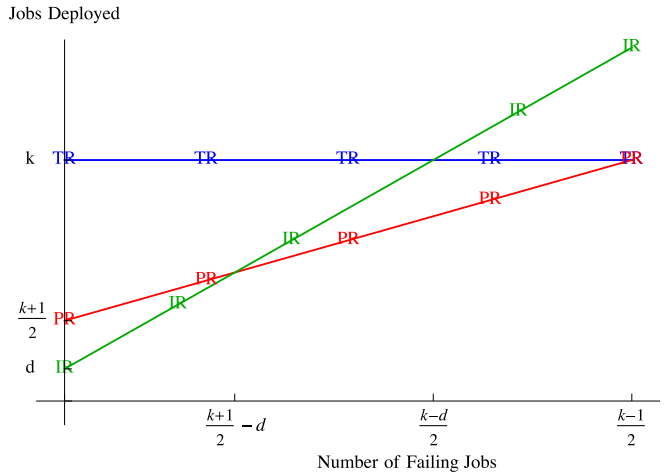


Fig. 8. Iterative redundancy (IR) injects more resources than progressive (PR) and traditional (TR) redundancy into the tasks with the most failing jobs. However, overall, IR uses fewer resources than PR and TR because IR saves resources on jobs with few failures.

node reliability; however, it is still not immune to drops in r, as the system reliability dips a fair amount. Iterative redundancy has the largest variations in jobs per task but keeps the overall reliability fairly constant.

Fig. 5 illustrates how iterative redundancy outperforms progressive redundancy in terms of cost factor. When node reliability peaks, progressive redundancy maxes out system reliability and produces 100 percent correct results, but it cannot reduce the jobs per task below $\frac{k+1}{2}$. Here, progressive redundancy is wasting effort by asking more nodes than are needed. Iterative redundancy, on the other hand, is able to reduce the jobs per task to as low as 2.

7.2 Rapidly Varying Reliability

Fig. 6 shows how a rapidly changing node reliability affects the cost factor and the system reliability, using each of the three redundancy techniques. The rapid drop in node reliability (top) makes iterative redundancy's self-adaptation even more evident. The cost (middle) increases quickly in response to the node reliability drop, and the overall system reliability (bottom) remains high. Meanwhile, progressive redundancy and traditional redundancy force the system reliability to drop significantly because they impose a limit on the amount of redundancy that can be injected into the system, even when more is needed.

Fig. 7 shows the aggregate amount of computation required by the three redundancy techniques in the scenario from Fig. 6. Despite requiring more computation and producing higher system reliability than the other techniques when the node reliability drops, iterative redundancy requires 2.5 and 1.3 times less overall computational effort than traditional and progressive redundancy, respectively.

7.3 Advantages of Reliability via Self-Adaptation

Fig. 2 showed theoretically (and Fig. 10 will verify empirically) that iterative redundancy reduces the number of resources needed to achieve a given system reliability. Iterative redundancy accomplishes this by being smart about deploying resources. Low-risk situations represent opportunities for savings, whereas high-risk situations may require extra resources to ensure reliability. Iterative redundancy's ability to adapt to each situation's risk allows it to be optimal in its resource use.

We further illustrate the advantages of iterative redundancy through an example. While traditional redundancy injects the same amount of redundancy regardless of the scenario, iterative redundancy automatically determines which situations require more or less redundancy. For example, suppose we employed 15-vote traditional redundancy to execute two tasks: A and B. We would then execute 15 A jobs and 15 B jobs, even if all A jobs returned the same result while only 8 B jobs returned one result and 7 returned the other. In the end, we would be very confident in the A result, but much less confident in the B result. Had we used iterative redundancy instead, the software system would have automatically determined, after executing just a few jobs, that the A jobs are achieving a higher reliability and would have spent more resources on the B jobs. In the end, we would have been equally confident in the A and B results.

To illustrate how iterative redundancy injects extra redundancy into "unlucky" situations with disproportionately many failures, consider how the behavior of each technique differs in a "lucky" (low-risk) situation in which nearly all jobs return agreeing results versus an "unlucky" (high-risk) situation in which some jobs return results that disagree with the majority. Fig. 8 shows, for each technique, the relationship between the amount of redundancy employed (i.e., the total number of jobs distributed) and the number of jobs that have returned a minority disagreeing result. While we have duplicated this graph using empirical experiments, Fig. 8 uses the most exact theoretical data and symbolic labels on the axes to make it more instructive. For traditional redundancy, the amount of redundancy is constant regardless of how many nodes disagree with the majority. For progressive redundancy, the total number of jobs distributed is equal to the number of disagreeing, minority nodes plus $\frac{k+1}{2}$ (the number of nodes in the majority). Finally, the iterative redundancy technique distributes a total number of jobs equal to twice the number of the disagreeing, minority nodes plus d. While

both iterative and progressive redundancy adapt by applying additional resources in situations with some failing jobs, iterative redundancy uses more runtime information to make better decisions and to guarantee optimal resource allocation. Further, this adaptation allows iterative redundancy to automatically keep the software system reliability nearly constant.

One interesting side effect of progressive redundancy being less adaptive than iterative redundancy is that in certain situations, progressive redundancy is more predictable in terms of bounds on the response time. Section 8.2 discusses scenarios that may make progressive redundancy preferable to iterative redundancy.

7.4 Predicting Node Reliability

Predicting the reliability of nodes is difficult and comes at a cost. We have already shown that it is not necessary to estimate r to use iterative redundancy. The user only needs to specify how much improvement is needed (or how high a cost in execution time is acceptable) and the algorithm uses the available resources to achieve the highest possible system reliability. However, in some circumstances, it may be possible to estimate the reliability of the node pool as a whole or the distinct reliability of different classes of nodes and jobs.

Numerous techniques, such as spot-checking of results, blacklisting, and computing node credibility [49], have been proposed as mechanisms to determine the reliability of nodes and utilize that information to improve software system reliability. For example, in an attempt to use node reliability knowledge, BOINC has recently added *adaptive replication*, which prevents replication of a task if a trusted node returns its result. However, these techniques have various shortcomings. For example, Byzantine faults cannot be reliably spot-checked, and malicious nodes can earn credibility and mislead schemes for rating credibility. Moreover, these techniques incur performance penalties of varying severity. For example, spot-checking requires distributing jobs to which the result is already known, while estimating node credibility requires storing and updating the past behavior of every node. In a large software system, these performance costs are non-trivial, meanwhile iterative redundancy has no such costs.

Although knowing r is not necessary to use iterative redundancy, knowing r can help calculate the reliability of the software systems employing the technique. An improved estimate of r will result in a more accurate calculation of system reliability. Fig. 9 demonstrates that iterative redundancy's performance is virtually identical whether it has correct estimates of r or over- or underestimates it significantly. Here, $r = 0.7$, but is overestimated as 0.8 and underestimated as 0.6. Because iterative redundancy uses the resources given to it optimally, the estimate of r is irrelevant: iterative redundancy produces as reliable a system as is possible given the available resources.

In some environments, it may be possible to get domain-specific resource reliability information. For example, some nodes may perform only one kind of job reliably, or some tasks' failures may be correlated with one another. It may be possible to leverage that information to both better compute the overall system reliability, and to further improve identifying risky situations that require more resources. We discuss some of these approaches in Sections 9.1 and 10. Note, however, that such techniques cannot deal with the Byzantine threat model we have assumed in our work.

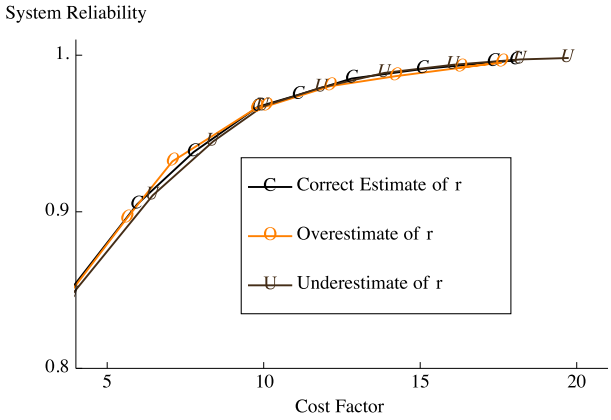


Fig. 9. The iterative redundancy algorithm is robust to poor estimates of r. Here, $r = 0.7$, but is overestimated as 0.8 and underestimated as 0.6.

8 Performance Evaluation

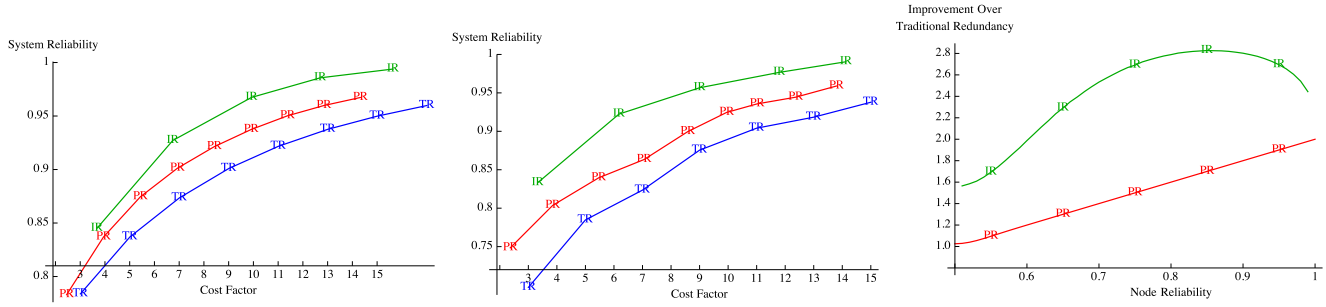
This section empirically compares the throughput and latency of traditional, progressive, and iterative redundancy.

8.1 Throughput

The throughput of a system employing a redundancy technique is inversely proportional to the number of jobs it has to deploy to compute each task, and therefore to the cost factor. Fig. 2 shows the theoretical predictions for the performance of the redundancy techniques. This section verifies those predictions with empirical data from, first, simulated systems executed in XDEVS and, then, from BOINC systems deployed on PlanetLab.

Fig. 10a shows empirical data from the XDEVS simulations that supports the claim that iterative redundancy outperforms traditional and progressive redundancy in the number of jobs and time to execute the computation. Each data point is the mean of 10,000 task executions. The data (for $r = 0.7$) closely agrees with our analytical predictions. The exact cost factor improvement of iterative redundancy depends on r. Fig. 10c demonstrates the improvement of iterative and progressive redundancy, as a function of r, over traditional redundancy. Progressive redundancy is most helpful for high r. If r is close to 0.5, the cost factor of k-vote progressive redundancy is close to k because, most likely, the nodes just barely reach the consensus. If, however, r is close to 1, progressive redundancy reaches the consensus quickly and shows greatest benefit over traditional redundancy. For r approaching 1, progressive redundancy uses 2.0 times fewer resources than traditional redundancy.

Iterative redundancy follows a similar trend. It is more efficient for larger r, but it is at least 1.6 times as efficient even for r close to 0.5. Iterative redundancy's efficiency peaks at 2.8 times that of traditional redundancy for $r \approx 0.86$. As r approaches 1, the efficiency of iterative redundancy decreases slightly, to ≈ 2.4 times that of traditional redundancy. We hypothesize that this decrease exists because, when almost all nodes are reporting correct results,



(a) XDEVS simulations (b) BOINC deployments (c) Improvement

Fig. 10. Experimental results from the (a) XDEVS simulations and (b) BOINC deployments, for $r = 0.7$. (c) The ratio improvement in cost factor for progressive (PR) and iterative (IR) redundancy over traditional redundancy varies with r.

utilizing runtime information to make redundancy decisions is somewhat less beneficial than when the nodes' behavior is highly variable. More precisely, as r increases, the cost $\mathbb{C}_R^k(r)$ to produce a constant increase in $\mathbb{R}_R^k(r)$ decreases linearly for traditional redundancy, but approaches a constant for iterative redundancy. We intend to conduct further experiments to test this hypothesis.

In our next set of experiments, we deployed the redundancy techniques on a BOINC system running on PlanetLab. Since we seeded some faults, we knew the reliability of the nodes would be no higher than $r = 0.7$. However, due to the other PlanetLab failures, we were unaware of the actual value of r. This scenario accurately represents typical real-world deployments. Fig. 10b depicts the system reliability as a function of the cost factor of each technique. Iterative redundancy, as we predicted, outperformed the other redundancy techniques, delivering the highest system reliability at the lowest cost in resources. Progressive redundancy also outperformed traditional redundancy.

The measurements in Fig. 10b allowed us to estimate the reliability of PlanetLab nodes. The executions consistently reported costs and system reliabilities consistent with $0.64 < r < 0.67$. Seeded faults lowered r to 0.7 and naturally occurring PlanetLab faults were responsible for the difference. The consistency of the derived node reliabilities, among multiple trials with different parameters and across all techniques, provides strong evidence for the validity of the experiments.

8.2 Latency

We have focused on minimizing the jobs needed to complete computations reliably. However, we have thus far ignored one aspect of iterative redundancy that may be important in some domains. Using traditional redundancy, a task server can deploy all k jobs at once. Meanwhile, using progressive or iterative redundancy, the task server must deploy several jobs and wait for the responses before possibly choosing to deploy more. Therefore, these techniques can increase the latency for a particular task. In the realm of DCAs, the number of tasks is far larger than the number of nodes, so the increased latency does not present a problem because the nodes can always execute jobs related to other tasks [4], [24]. In other words, no node will ever be idle and all nodes processing capability will be fully utilized. However, some applications may pose requirements on the latency for particular tasks.

A task server employing traditional redundancy attempts to start all the jobs related to a single task at once, in a single wave. In contrast, a task server employing progressive redundancy may wait for several waves of jobs to finish before deploying more; however, it guarantees that there will be no more than $\frac{k+1}{2}$ such waves. Iterative redundancy makes no such guarantees, and while it is very unlikely, any one task may require arbitrarily many waves of jobs.

Fig. 11 shows the average latency for tasks using the three redundancy techniques, as measured in XDEVS simulations. The response time depends on the cost factor. For the instances measured, progressive redundancy took between 1.4 and 2.5 times longer and iterative redundancy between 1.4 and 2.8 times longer to respond than traditional redundancy. Thus, progressive redundancy offers a lower average latency and a lower upper bound on latency than iterative redundancy, making progressive redundancy more predictable and better suited for some types of software systems.

In addition to response time, some domains concerned with privacy may want a hard limit on the number of times a task may be replicated and deployed. Traditional and progressive redundancy can provide such limits, while iterative redundancy cannot. However, it is possible to impose an artificial limit on the number of replicas. While we have not yet fully investigated this option, our intuition is that such a limit would reduce both the system reliability and the cost factor. However, these effects would be minor because they

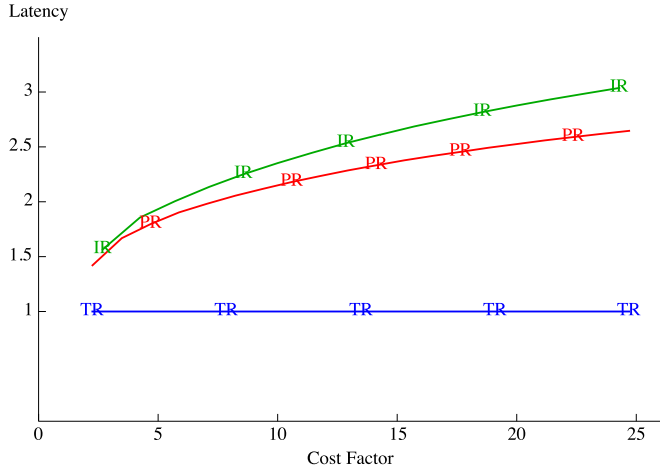


Fig. 11. The average latency for tasks using traditional (TR), progressive (PR), and iterative (IR) redundancy.

only affect the low-probability cases that use a large number of replicas.

The idea of artificially limiting the number of replicas highlights an interesting research direction. In this paper, we have identified a new dimension along which redundancy techniques can vary: the degree to which runtime information about the reliability of individual tasks is leveraged to increase or decrease replication of those tasks on-the-fly. This dimension captures a trade-off between the response time for each individual task and the performance of the software system as a whole. Viewed from this dimension, traditional redundancy techniques represent the extreme of this spectrum where individual task response time is minimized but performance of the system as a whole is suboptimal. Iterative redundancy represents the other extreme of the spectrum and progressive redundancy lies within the spectrum. Iterative redundancy optimizes the performance of the system as a whole while increasing the response time of individual tasks. Other techniques along this spectrum (such as iterative redundancy with an artificial limit on the number of replicas) may provide interesting insights into the trade-off and remain the focus of our future work.

9 Relaxing Assumptions

We made several assumptions, listed in Section 2.3, that helped to clarify how and why iterative redundancy works. We assumed that every job sent to the node pool had the same probability of failure, that those failures were independent, and that the result of every job was one of two possible values. This section explains how redundancy can apply to DCAs deployed on networks without these assumptions, and, in some cases, can even benefit from their relaxation. Note that our final assumption, that the reliability of the client that receives the final result is beyond the scope of our analysis, does not make sense to relax because the software system can never be more reliable than the node submitting the original computation. It is feasible to replicate the client, but we leave this possibility for future work.

9.1 Probability Distribution

Equations (1) through (6), as well as the analysis in Section 6, reflect the assumption that each job has an equal, and independent probability of failure. We made this assumption based on the fact that many DCAs (e.g., BOINC [4] and Hadoop [31]) assign jobs to nodes from the node pool at random; therefore, from the node reliability perspective, every job submitted to the job queue has the same probability of failure. However, for some other types of software systems, this assumption might not hold. Most notably, when independent groups of software developers write software for the same specifications, their products' errors are often correlated [37]. In these cases, and if the correlation is known, the only necessary change to Equations (1) through (6) is the replacement of r with appropriate reliabilities of the relevant nodes. For example, if r_c denoted the reliability of a particular job c, Equation (3) becomes

$$\mathbb{C}^k_{PR} = \frac{k+1}{2} + \sum_{i=\frac{k+3}{2}}^{k} \sum_{j=i-\frac{k+1}{2}}^{\frac{k-1}{2}} \binom{i-1}{j} \prod_{c=1}^{j} r_c \prod_{c=j+1}^{i} (1-r_c).$$

The final cost and probability of failure would then depend on the probability distribution.

We have so far assumed that job failures are independent. However, in some cases, probabilities of job failures may depend on each other: e.g., if a node in one part of the world fails because of a natural disaster, others near it are more likely to fail as well. If the dependencies among job failure probabilities are known, job schedulers can use the additional information to decrease the probability of failure, using a scheme based on the complex form of the iterative redundancy algorithm or credibility-based fault tolerance [49]. However, if the dependencies are unknown, iterative redundancy can still be used. The analysis of the algorithm would again change as above, with r being replaced with the specific reliabilities of the relevant nodes.

This statement opens a number of questions, such as whether there exists an optimal distribution algorithm to minimize both the cost and probability of failure. We foresee, however, a balance between cost and reliability. One example that leads us to this hypothesis is that following the naïve algorithm of asking the most reliable nodes first would likely minimize the probability of failure, but increase the cost because the reliable nodes would be overworked.

9.2 Local Information

We had assumed that every node on the network has the same knowledge about the reliability of the network. In real-world software systems, it is more likely that every node has intimate knowledge about its local neighbors and relatively little knowledge about distant portions of the network. Further, during the course of computation, each node may collect information about other nodes it uses for subcomputations, such as the frequency of disagreement with others, thus generating information that is not available globally. Since in our approach, every distribution decision is made locally, each node can use the most accurate information available. Depending on the kind and amount of information that nodes can collect at runtime, it may be possible to reduce the probability of failure and expected cost of progressive redundancy, though it remains future work to explore both the information-collection algorithms and the most effective uses of that information.

9.3 Non-Binary Results

The assumption that the result of every task is a single bit, as in decision NP-complete problems, has simplified our analysis thus far, but it actually turns out to be the worst-case scenario. Compare two types of tasks: the first asks whether $2^2 = 4$ and the second asks for the result of 2^2. For the first task, all nodes that fail and report the wrong result will report "no", possibly making it difficult to distinguish between the correct and incorrect result. For the second task, nodes may report distinct integers, and it may be possible to determine that the correct result is 4 even if more than half of the nodes fail, because the plurality (though not the majority) will report the correct result.

Iterative redundancy is naturally applicable to software systems that perform tasks with non-binary results. The probabilities of failure and costs of execution we have

presented are upper bounds for non-binary systems, and all our analysis applies as is. For all (binary and non-binary) systems with malicious nodes that collude to try to cause failures, our analysis gives tight bounds on the failure probabilities and execution costs. It is possible to develop a threat model that is weaker than ours and analyze non-binary systems that disallow cooperation between malicious nodes; however, such an analysis is unlikely to produce meaningful improvements on the bounds we present.

Another important aspect of non-binary results is that two non-identical results may actually represent the same information (e.g., evaluations of $\sqrt{2}$ may return slight differences in the least significant bits). In such cases, the comparison of jobs' results is problem-specific, and the distributing nodes must be equipped with the proper comparison algorithms. BOINC uses homogeneous redundancy, an approach that sorts nodes into equivalence classes that report identical answers, to resolve this issue.

10 Related Work

While our analysis of iterative redundancy assumed an independence of failures, the technique also benefits software systems for correlated failures, such as those typically produced by independent groups of developers writing software to the same specifications, known as n-version programming [37]. In the limit, if all software components always fail in identical ways, redundancy cannot help improve reliability. Section 9.1 discussed how iterative redundancy analysis would change if the correlation between faults were known.

We based progressive redundancy on a self-configuring optimistic programming technique [12], [13] aimed at component-based systems. Such systems allow for asynchronous job scheduling; however, they focus on minimizing response time and typically allocate finite resources to each task. DCAs relax these limits, which allows deploying jobs without a priori knowledge of node reliability or a bound on the number of jobs. As part of our experimental evaluation described in Section 8, we have adapted self-configuring optimistic programming to apply to DCAs, implemented these ideas as progressive redundancy, and deployed progressive redundancy both in simulation and on BOINC deployments.

Primary backup [19] and active replication [50] are two popular redundancy architectures. Primary backup uses multiple servers to improve the reliability of a service—one server designated as primary, while the others act as backups. The primary-backup architecture handles on-the-fly updates of the backups to ensure limits on losses from primary-server failures, while keeping the cost of updates among the servers low. Primary backup is widely used in commercial fault-tolerant systems [19]. Iterative redundancy complements primary backup by specifying, at runtime, how many backups should exist to guarantee the maximum reliability for a given cost.

Active replication removes the centralized control of primary backup and minimizes losses that occur when some replicas fail. Active replication incurs a high cost associated with keeping all replicas synchronized [50]. Again, iterative redundancy complements active replication by specifying, at runtime, how many replicas should exist to guarantee a particular level or reliability. While primary backup and active replication propose mechanisms for implementing redundancy in distributed systems, iterative redundancy improves the efficiency of those mechanisms.

ZZ [53] applies the idea of using runtime information to improve system reliability when faced with Byzantine fault in service-based computing. ZZ deploys component replicas based on a runtime-reactive algorithm to reduce the number of necessary replicas to tolerate f failures to fewer than $2f + 1$. Some of the underlying ideas in ZZ are similar to progressive redundancy, though the goal is somewhat different.

Credibility-based fault tolerance [49] uses probability estimates to efficiently detect erroneous results submitted by malicious volunteers in volunteer-computing systems. The probability calculations used by credibility-based fault tolerance resemble the complex form of the iterative redundancy algorithm. However, credibility-based fault tolerance does not incorporate our simplifying insight that allows the algorithm to function without any estimates of node reliability. As a result, credibility-based fault tolerance is forced to rely on spot-checking with blacklisting. However, Byzantine faults cannot be reliably spot-checked, and malicious nodes can earn credibility and fool schemes for rating credibility.

Hwang and Kesselman [32] proposed a method for injecting fault tolerance into grids that handles a wide variety of faults within distributed systems. This work uses a service to detect crash failures (and an extension to allow the system designer to specify how to detect other failures) and a failure-handling framework that enforces designer-defined policies [32].

Traditional checkpoint techniques can also be applied to DCAs to log partially completed work and prevent data and computation loss in cases of crash failures. Checkpoints can be effective when individual subcomputations take a long time to complete [48]. Further, using checkpoints and replication together can reduce the number of replicas needed to detect Byzantine failures [2] over what the standard Byzantine agreement protocols [50] require.

Autonomous agents capable of detecting failing components and initiating on-demand replication allow autonomic fault tolerance, although the developer has to implement fault-specific detection mechanisms into these agents [23]. Nevertheless, this work is a step in the right direction, as Internet-sized systems' complexity does not allow for centralized managers, and thus these systems must manage themselves.

Runtime information has been used to reduce the resource requirements in crowdsourcing systems, such as AutoMan [7]. In several ways, this technique is similar to iterative redundancy, though it requires a random attack model and does not apply to the Byzantine threat model iterative redundancy handles. AutoMan uses the null hypothesis test as a measure of reliability, which makes it more efficient than iterative redundancy. If applied to environments with random (and not Byzantine) failures, iterative redundancy can be made similarly efficient.

Iterative redundancy is applicable to a wide variety of DCAs, such as the Globus grid middleware [30], MapReduce [24], the organic grid [21], sTile [16], [17],

distributed robotics systems (e.g., [18], [45]), and @home and BOINC systems [4], [5], [11], [39], [43].

The Globus grid middleware [30] is a widely used DCA. Globus includes a fault-detecting service that can handle node and network-link crashes. This service allows the detection of component failures and component replication to restore functionality. However, this detection is typically expensive and is applicable to heavyweight components [35]. Similarly, in systems with components capable of reporting their own failures, or with easily detectable failures, component replication can ensure sustainability and other qualities of service [46].

MapReduce [24] is another well-known DCA that is applicable to certain problems that manipulate massive datasets. In MapReduce systems, the engineer designs two functions: `Map` and `Reduce`. The `Map` function takes a computation and divides it into a small number of tasks that can be solved in parallel. The `Reduce` function takes the solutions to several tasks and combines them into a solution to the original problem. The MapReduce infrastructure handles taking a single computation, distributing it, and combining the solutions into the final result. The best known use of MapReduce is computing Google's PageRank [24]. At its core, that MapReduce infrastructure does not use redundancy; however, Hadoop [31], a popular implementation of MapReduce, uses traditional redundancy in its file system to reliably store data. While Hadoop can redeploy failed jobs, its support for redundancy is rudimentary at best.

The organic grid [21] is a DCA that is used to solve easily parallelizable problems, such as NP-complete problems, and problems susceptible to dynamic programming, such as nucleotide-nucleotide alignment and matrix multiplication. The organic grid decomposes computational tasks into subtasks and assigns each subtask to a mobile agent, whose job is to autonomously locate a node with adequate resources to perform the subcomputation. The organic grid can tolerate nodes in the middle of the tree becoming unresponsive, but currently, the organic grid offers no fault-tolerance mechanism for incorrect subcomputation results, or even corrupted or lost results due to network communication failures, which we believe is a significant reason for its lack of adoption in industry. However, since the organic grid deals with predominantly nonblocking subcomputations and employs a decentralized scheduler, it is susceptible to our iterative redundancy technique.

sTile [16], [17] solves NP-complete problems, such as 3-SAT, determining the structure of proteins, and optimally allocating resources. sTile decomposes NP-complete problems into the smallest possible subcomputations (on the order of complexity of simple two-input binary gates) and distributes those subcomputations onto a network. The subcomputations are iterative, so the result of each subcomputation is one or more subcomputations that are then distributed onto other nodes. As the nodes perform and distribute the subcomputations, they provide an autonomic decentralized distribution service. Note that the nature of the problems sTile tackles, NP-complete problems, is integral to the decomposition into iterative subcomputations. Current implementations of sTile rely on trustworthy nodes and do not employ fault tolerance. Proposals for making sTile fault and adversary tolerant have argued that sTile is susceptible to both traditional redundancy and biologically-inspired fault-tolerance techniques [15]. The redundancy technique we defined in this paper build on the existing proposed ideas and are more efficient than the technique proposed in [15].

11 Contributions

We presented iterative redundancy, a novel method for designing and implementing distributed software systems with an automated, self-adaptive, efficient technique for improving system reliability. Iterative redundancy is more efficient than existing methods in its use of resources; in fact, it guarantees optimal resource use. Iterative redundancy is self-adaptive because it (1) automatically detects when resource reliability drops and injects extra redundancy to counter that drop, (2) automatically identifies "unlucky" parts of the computation that happen to deploy on disproportionately many compromised resources and expends more resources to increase the reliability of those parts, and (3) does not rely on *a priori* estimates of resource reliability. In addition to a rigorous theoretical analysis, we verified iterative redundancy's self-adaptivity and efficiency with an empirical evaluation based on two deployments: the XDEVS discrete event simulator and the BOINC volunteer-computing system. Our empirical results support our theoretical findings and the deployment of our technique on a real-world software system.

Iterative redundancy serves as one extreme on a spectrum of redundancy techniques with different trade-offs and benefits. While providing a concrete improvement on the state-of-the-art, our work also serves as a starting point toward a further exploration of how runtime information can be used to improve software system reliability.

Acknowledgments

This work has been supported by the US Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-11-C-4021, IARPA under Contract No. N66001-13-1-2006, and the National Science Foundation under award numbers CCF-1117593, CCF-1218115, and CCF-1321141. The work has also been supported in part by Infosys Ltd. Yuriy Brun is the corresponding author.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proc. ACM Symp. Oper. Syst. Principles*, Brighton, U.K., 2005, pp. 59–74.

[2] A. Agbaria and R. Friedman, "A replication- and checkpoint-based approach for anomaly-based intrusion detection and recovery," in *Proc. Int. Workshop Secur. Distrib. Comput. Syst.*, 2005, pp. 137–143.

[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "Consensus with Byzantine failures and little system synchrony," in *Proc. Int. Conf. Dependable Syst. Netw.*, Philadelphia, PA, USA, 2006, pp. 147–155.

[4] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proc. IEEE/ACM Int. Workshop Grid Comput.*, Pittsburgh, PA, USA, 2004, pp. 4–10.

[5] D. P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer, "SETI@home: An experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[6] J. Andrade, L. Berglund, M. Uhlén, and J. Odeberg, "Using grid technology for computationally intensive applied bioinformatics analyses," *In Silico Biol.*, vol. 6, no. 6, pp. 495–504, 2006.

[7] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor, "AutoMan: A platform for integrating human-based and digital computation," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Language Appl.*, Tucson, AZ, USA, 2012, pp. 639–654.

[8] D. W. Barowy, D. Gochev, and E. D. Berger, "Data debugging," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Language Appl.*, Portland, OR, USA, 2014, pp. 507–523.

[9] S. A. Baset and H. Schulzrinne, "An analysis of the skype peer-to-peer Internet telephony protocol," in *Proc. IEEE Conf. Comput. Commun.*, Barcelona, Spain, Apr. 2006, pp. 1–11.

[10] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich, "Soylent: A word processor with a crowd inside," in *Proc. ACM Symp. User Interface Softw. Technol.*, New York, NY, USA, 2010, pp. 313–322.

[11] BOINC. (2009). The Berkeley open infrastructure for network computing [Online]. Available: http://boinc.berkeley.edu

[12] 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," *J. Syst. Archit.*, vol. 47, no. 9, pp. 763–781, 2002.

[13] A. Bondavalli, F. Di Giandomenico, and Jie Xu, "A cost-effective and flexible scheme for software fault tolerance," *J. Comput. Syst. Sci. Eng.*, vol. 8, no. 4, pp. 234–244, 1993.

[14] Y. Brun, G. Edwards, J. young Bang, and N. Medvidovic, "Smart redundancy for distributed computation," in *Proc. 31st Int. Conf. Distributed Comput. Syst.*, Minneapolis, MN, USA, Jun. 2011, pp. 665–676.

[15] Y. Brun and N. Medvidovic, "Fault and adversary tolerance as an emergent property of distributed systems' software architectures," in *Proc. 2nd Int. Workshop Eng. Fault Tolerant Syst.*, Dubrovnik, Croatia, Sep. 2007, pp. 38–43.

[16] Y. Brun and N. Medvidovic, "Keeping data private while computing in the cloud," in *Proc. 5th Int. Conf. Cloud Comput.*, Honolulu, HI, USA, Jun. 2012, pp. 285–294.

[17] Y. Brun and N. Medvidovic, "Entrusting private computation and data to untrusted networks," *IEEE Trans. Dependable Secure Comput.*, vol. 10, no. 4, pp. 225–238, Jul./Aug. 2013.

[18] Y. Brun and D. Reishus, "Path finding in the tile assembly model," *Theoretical Comput. Sci.*, vol. 410, no. 15, pp. 1461–1472, Apr. 2009.

[19] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems*, 2 ed. New York, NY, USA: ACM Press/Addison-Wesley, 1993, pp. 199–216.

[20] R. Buyya, S. Date, Y. M.-Matsumoto, S. Venugopal, and D. Abramson, "Neuroscience instrumentation and distributed analysis of brain activity data: A case for eScience on global grids," *Concurrency Comput.: Practice Exp.*, vol. 17, no. 15, pp. 1783–1798, 2005.

[21] A. J. Chakravarti and G. Baumgartner, "The organic grid: Self-organizing computation on a peer-to-peer network," in *Proc. Int. Conf. Autonomic Comput.*, New York, NY, USA, 2004, pp. 96–103.

[22] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Proc. Int. Workshop Design Issues Anonymity Unobservability*, 2001, pp. 46–66.

[23] A. D. L. Almeida, J.-P. Briot, S. Aknine, Z. Guessoum, and O. Marin, "Towards autonomic fault-tolerant multi-agent systems," presented at the Latin American Autonomic Computing Symp., Petropolis, RJ, Brazil, 2007.

[24] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," presented at the Symp. Operating System Design Implementation, San Francisco, CA, USA, Dec. 2004.

[25] J. Deng, O. Russakovsky, J. Krause, M. S. Bernstein, A. Berg, and L. Fei-Fei, "Scalable multi-label annotation," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, Toronto, ON, Canada, 2014, pp. 3099–3102.

[26] J. Duerig, R. Ricci, J. Zhang, D. Gebhardt, S. Kasera, and J. Lepreau, "Flexlab: A realistic, controlled, and friendly environment for evaluating networked systems," in *Proc. Workshop Hot Topics Netw.*, Irvine, CA, USA, Nov. 2006, pp. 103–108.

[27] G. Edwards and N. Medvidovic, "A highly extensible simulation framework for domain-specific architectures," Center for Software Engineering, Univ. Southern California, Los Angeles, CA, USA, Tech. Rep. USC-CSSE-2009-511, 2009.

[28] M. D. Ernst, R. Just, S. Millstein, W. M. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative verification of information flow for a high-assurance app store," in *Proc. ACM Conf. Comput. Commun. Secur.*, Scottsdale, AZ, USA, Nov. 2014, pp. 1092–1104.

[29] A. D. Friedman and P. R. Menon, *Fault Detection Digital Circuits*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1971.

[30] The Globus alliance. (2005) [Online]. Available: http://www.globus.org

[31] Hadoop. (2009) [Online]. Available: http://hadoop.apache.org

[32] S. Hwang and C. Kesselman, "A flexible framework for fault tolerance in the grid," *J. Grid Comput.*, vol. 1, no. 3, pp. 251–272, Sep. 2003.

[33] P. Jalote, *Fault Tolerance in Distributed Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1994.

[34] S. R. Jeffery, M. J. Franklin, and M. Garofalakis, "An adaptive RFID middleware for supporting metaphysical data independence," *VLDB J.*, vol. 17, no. 2, pp. 265–289, Mar. 2008.

[35] H. Jin, D. Zou, H. Chen, J. Sun, and S. Wu, "Fault-tolerant grid architecture and practice," *J. Comput. Sci. Technol.*, vol. 18, no. 4, pp. 423–433, 2003.

[36] T. Kimoto, K. Asakawa, M. Yoda, and M. Takeoka, "Stock market prediction system with modular neural networks," in *Proc. Int. Joint Conf. Neural Netw.*, Jun. 1990, pp. 1–6.

[37] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Softw. Eng.*, vol. 12, no. 1, pp. 96–109, Jan. 1986.

[38] I. Koren and C. Mani Krishna, *Fault-Tolerant Systems*. Amsterdam, The Netherlands: Elsvier, 2007.

[39] E. Korpela, D. Werthimer, D. Anderson, J. f. Cobb, and M. Lebofsky, "SETI@home—Massively distributed computing for SETI," *IEEE MultiMedia*, vol. 3, no. 1, pp. 78–83, Jan. 1996.

[40] M. Lamanna, "The LHC computing grid project at CERN," *Nuclear Instrum. Methods Phys. Res. Section A: Accelerators, Spectrometers, Detectors Assoc. Equip.*, vol. 534, no. 1/2, pp. 1–6, 2004.

[41] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Languages Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[42] C. B. Laney, *Computational Gasdynamics*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[43] S. M. Larson, C. D. Snow, M. R. Shirts, and V. S. Pande, *Folding@-Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable Problems in Computational Biology*. New York, NY, USA: Horizon, 2002.

[44] D. Long, A. Muir, and R. Golding, "A longitudinal survey of Internet host reliability," in *Proc. Symp. Reliable Distrib. Syst.*, Bad Neuenahr, Germany, 1995, pp. 2–9.

[45] N. Medvidovic, H. Tajalli, J. Garcia, Y. Brun, I. Krka, and G. Edwards, "Engineering heterogeneous robotics systems: A software architecture-based approach," *IEEE Comput.*, vol. 44, no. 5, pp. 61–71, May 2011.

[46] A. Nguyen-Tuong, "Integrating fault-tolerance techniques in grid applications," Ph.D. dissertation, Dept. Comput. Sci., Univ. Virginia, Charlottesville, VA, USA, 2000.

[47] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the Internet," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 59–64, 2003.

[48] S. B. Priya, M. Prakash, and K. K. Dhawan, "Fault tolerance-genetic algorithm for grid task scheduling using check point," in *Proc. Int. Conf. Grid Cooperative Comput.*, 2007, pp. 676–680.

[49] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," *Future Gener. Comput. Syst.*, vol. 18, no. 4, pp. 561–572, 2002.

[50] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[51] A. Setiawan, D. Adiutama, J. Liman, A. Luther, and R. Buyya, "GridCrypt: High performance symmetric key using enterprise grids," in *Proc. Int. Conf. Parallel Distributed Comput., Appl. Technol.*, Singapore, 2004, pp. 872–877.

[52] M. Sipser, *Introduction to the Theory of Computation*. Boston, MA, USA: PWS Publishing Company, 1997.

[53] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "ZZ and the art of practical BFT execution," in *Proc. Eur. Conf. Comput. Syst.*, Salzburg, Austria, 2011, pp. 123–138.

Yuriy Brun received the MEng degree from the Massachusetts Institute of Technology in 2003 and the PhD degree from the University of Southern California in 2008. He is an assistant professor in the School of Computer Science, University of Massachusetts, Amherst. He completed his postdoctoral work in 2012 at the University of Washington, as a CI fellow. His research focuses on software engineering, distributed systems, and self-adaptation. He received the US National Science Foundation (NSF) CAREER Award in 2015, a Microsoft Research Software Engineering Innovation Foundation Award in 2014, and an IEEE TCSC Young Achiever in Scalable Computing Award in 2013. He is a member of the IEEE, the ACM, and the ACM SIGSOFT. More information is available on his homepage: http://www.cs.umass.edu/ brun/.

Jae young Bang received the MS degree from the University of Southern California. He is currently working toward the PhD degree in the Computer Science Department, University of Southern California. He joined the PhD program as a USC Annenberg graduate fellow in 2010. His research interest spans from collaborative software design and development to large and distributed software systems. He is a member of the IEEE, ACM, and ACM SIGSOFT. More information is available on his homepage: http://ronia.net/.

George Edwards received the BS degree from Vanderbilt University, and the MS and PhD degrees from USC, all in computer science. He is a part-time lecturer in the Computer Science Department, University of Southern California, where he teaches undergraduate-level programming and graduate-level software engineering classes. He also provides consulting and expert witness services to companies involved in software-related litigation. More information is available on his homepage: http://softarch.usc.edu/~gedwards/.

Nenad Medvidovic received the PhD degree in 1999 from the University of California, Irvine. He is a professor in the Computer Science Department, University of Southern California. His research focuses on the software architectures of large, distributed, mobile, and embedded systems. He is a senior member of the IEEE, and a member of the ACM, and ACM SIGSOFT. More information is available on his homepage: http://sunset.usc.edu/ neno/.