

Smart Redundancy for Distributed Computation

Yuriy Brun
University of Washington
Seattle, WA, 98195-2350, USA
brun@cs.washington.edu

George Edwards, Jae young Bang, and Nenad Medvidovic
University of Southern California
Los Angeles, CA 90089-0781, USA
{gedwards, jaeyounb, neno}@usc.edu

Abstract

Many distributed software systems allow participation by large numbers of untrusted, potentially faulty components on an open network. As faults are inevitable in this setting, these systems utilize redundancy and replication to achieve fault tolerance. In this paper, we present a novel “smart” redundancy technique called iterative redundancy, which ensures efficient replication of computation and data given finite processing and storage resources. Iterative redundancy is more efficient and more adaptive than comparable state-of-the-art techniques. We show how systems that solve computational problems using a network of independent nodes can benefit from iterative redundancy. We present formal analytical and empirical analyses, demonstrating iterative redundancy on a real-world volunteer computing system and comparing it to existing methods.

1. Introduction

Many software systems today, such as distributed data stores (e.g., Freestore) and peer-to-peer A/V streaming applications (e.g., Skype), consist of large numbers of autonomous software and hardware participants that interact over untrusted networks. These systems utilize redundancy mechanisms to tolerate faults and achieve acceptable levels of reliability. In this paper, we focus on one subset of these systems: *distributed computation architectures*, which solve massive problems by deploying highly parallelizable computations (i.e., sets of independent tasks) to dynamic networks of potentially faulty and untrusted computing nodes. Widely known and successful distributed computation architectures include grid systems (e.g., Globus [15]), volunteer computing systems (e.g., BOINC [5]), and MapReduce systems (e.g., Hadoop [16]). Distributed computation architectures are used extensively for diverse applications, including cryptanalysis [25], web analytics [11], and scientific simulations in fields such as physics [21], bioinformatics [4], and economics [19].

It is imperative that distributed computation architectures be able 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 system or compromise other participants. Today’s distributed computation architectures aim to ensure the correct execution of each task through what we term *traditional redundancy*: multiple independent machines perform the same computation and their results are checked for agreement. This technique is costly, however, as replicating each task n times requires expending a factor of n resources or suffering a factor of n slowdown in performance.

In this paper, we propose a new software redundancy technique — *iterative redundancy* — that exploits the properties of distributed computation architectures to adapt to changing execution environments and improve reliability more efficiently than existing alternatives. More generally, iterative redundancy is applicable to systems that perform computations using a pool of independent processing resources and have the ability to monitor the resources and make task-deployment decisions at runtime. We describe iterative redundancy, formally analyze its cost and performance impacts, and perform a rigorous empirical evaluation on a real-world volunteer computing system.

We compare iterative redundancy against two alternatives: in addition to traditional redundancy, for comparison purposes we have adapted a related technique from the area of self-configuring optimistic programming research [7, 6], which we refer to as *progressive redundancy*. Iterative redundancy outperforms both these alternatives by leveraging runtime information to inject redundancy where it is essential and eliminate it where it is unnecessary. We demonstrate two key characteristics that make iterative redundancy superior to both traditional and progressive redundancy: efficiency and self-adaptivity. Iterative redundancy is more *efficient* than both these alternative methods because it produces the same level of system reliability at a lower cost in employed system 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 system reliability. Further, iterative redundancy is *self-adaptive* because it recognizes the situations in which a computation is at a high risk of failure and injects additional redundancy to mitigate that risk.

The remainder of this paper is organized as follows. Section 2 presents the definitions and assumptions underlying our work. Section 3 describes the three redundancy techniques and provides their theoretical analysis, while Section 4 presents the empirical evaluation. Section 5 discusses our results. The paper is concluded by an overview of the related work and a recap of our contributions.

2. Definitions and Assumptions

In this section, we define our model of a distributed computation architecture, state the threat model we will consider, and enumerate 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 distributed computation architecture. A *task* is one of the parts 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 distributed computation architecture, 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.

This system model is depicted in Figure 1. The model accurately represents a number of distributed computation architectures, including the BOINC family of volunteer computing systems [3, 5].

2.2. Threat Model

In this paper, we employ the Byzantine failure model, which is the most general and widely accepted threat model [14, 18, 20] that has been applied to numerous distributed systems [1, 2, 18]. The threat model includes Byzantine failures and allows for malicious nodes that collude and form cartels to try to mislead and break computations. There are two important statements to be made about this threat model:

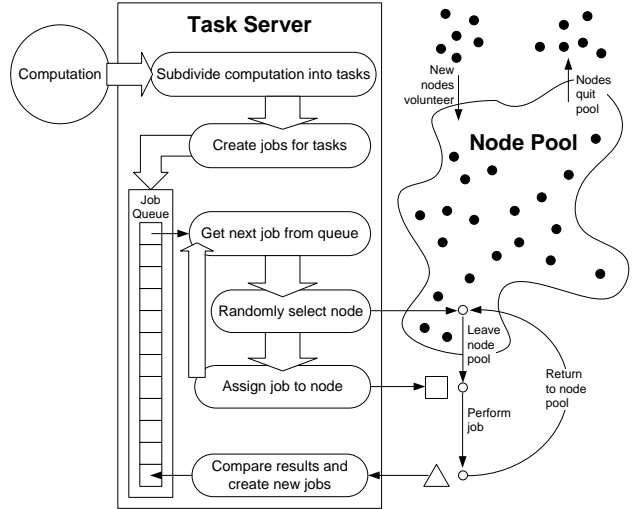


Figure 1. A model of a computational grid.

1. Our threat model is at least as strong as those used by redundancy techniques currently deployed in distributed computation architectures [3, 9, 11]. 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 system reliability. On the other hand, we make no assumptions that existing implementations of distributed computation architectures do not make.

2. 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 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.

2.3. Assumptions

In this section, we state four assumptions about the nodes of the network on which a distributed computation architecture is deployed. These assumptions simplify the description and analysis of the three redundancy techniques. In Section 5.3, we relax these assumptions and demonstrate that iterative redundancy still applies and, in some cases, performs even better on more general networks.

1. Every job sent to the node pool has the same probability of failure. While some nodes may be more reliable than others, the jobs are assigned to the nodes at random.

2. Node failures are independent of each other.

3. The result of every job is one of two possible values (e.g., “yes” or “no,” as in NP-complete problems [26]).

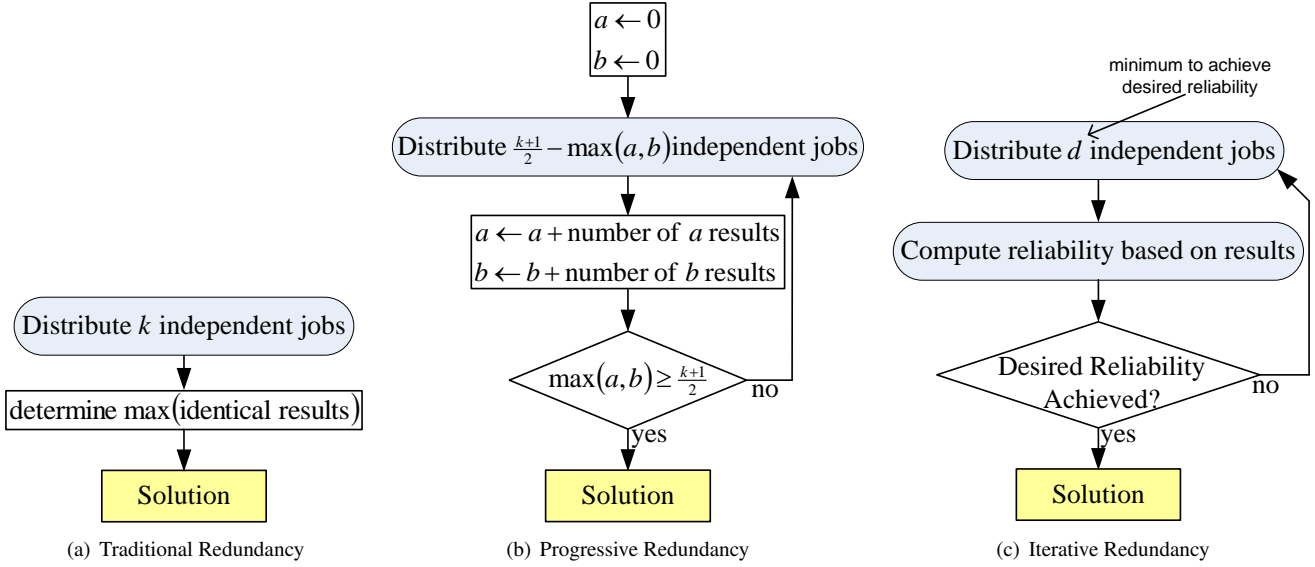


Figure 2. Schematics of traditional (a), progressive (b), and iterative (c) redundancy techniques.

This assumption is quite common in fault-tolerance research [20]. 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.

4. The reliability of the client that receives the final result of the computation is excluded from the system’s reliability.

3. Redundancy Algorithms

In this section, we specify three redundancy techniques: the state-of-the-practice traditional redundancy, the state-of-the-art progressive redundancy, and our novel iterative redundancy. To characterize the behavior of each technique, we derive formulae for two measures of their effect on systems: the *system reliability* $\mathbb{R}(r)$ achieved by and the *cost factor* $\mathbb{C}(r)$ of applying the redundancy technique. Both of these measures are functions of the average reliability $r \in [0, 1]$ of the node pool; r can be defined as the fraction of time a job returns the correct response. For completeness, we present the somewhat complex formulae for cost and reliability of each technique. As an aid to the reader, Figure 3 provides a graphical depiction of the costs and reliabilities. Further, in Section 4, we verify the formulae’s correctness experimentally. In Section 3.1, we describe traditional redundancy. In Section 3.2 we describe progressive redundancy, which is an adaptation of techniques from the field of self-configuring optimistic programming to the realm of distributed computation architectures. Finally, in Section 3.3, we describe our iterative redundancy technique.

3.1. Traditional Redundancy

The *k-vote traditional redundancy* technique (sometimes called *k-modular redundancy* [20]) performs k independent executions (where $k \in \{3, 5, 7, \dots\}$) 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. Traditional redundancy is easy to implement. Modern implementations of distributed computation architectures, including BOINC [3, 5] and Hadoop [16], rely on traditional redundancy. Figure 2(a) graphically depicts the traditional redundancy algorithm.

Example: Suppose $k = 19$ and each node’s reliability is $r = 0.7$. Distributing a single job for each task (i.e., not using any redundancy) results in a system reliability of 0.7. Using traditional redundancy 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 $\mathbb{R}_{TR}^k(r)$ and the cost factor as $\mathbb{C}_{TR}^k(r)$. Traditional k -vote redundancy repeats every task k times, independently of r . Thus,

$$\mathbb{C}_{TR}^k(r) = k. \quad (1)$$

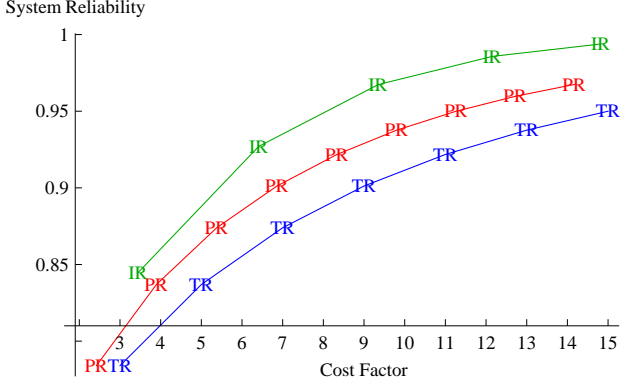


Figure 3. The reliability of a system approaches 1 exponentially, as a function of cost, for traditional, progressive, and iterative redundancy techniques (here, $r = 0.7$).

The reliability of a system with k -vote traditional redundancy is the probability that at least a consensus of jobs ($\frac{k+1}{2}$) does not fail, in other words, the sum of the probabilities that only 0, 1, \dots , and $\frac{k-1}{2}$ jobs fail. Thus,

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

Figure 3 graphs the system reliability vs. the cost factor of redundancy techniques for a node pool of reliability $r = 0.7$. The reliability of a system employing traditional redundancy (labeled “TR”) approaches 1 exponentially as the cost factor grows linearly.

3.2. Progressive Redundancy

The field of self-configuring optimistic programming studies techniques for improving the efficiency of redundant systems. Typically, self-configuring optimistic programming schemes are aimed at component-based software. As part of our research into the state-of-the-art in redundancy techniques, we observed that they can be redesigned to apply to distributed computing architectures. We have indeed leveraged one such scheme [7] to develop *progressive redundancy*. While, to our knowledge, progressive redundancy is not used today in any deployed distributed systems (in fact, we believe the algorithm we present in this section to be the first one of its kind), we introduce it here because, in a sense, it represents a “mid-way” point between traditional and iterative redundancy, both in terms of the achieved system reliability as a function of cost, and in helping to better explain iterative redundancy to a reader who is familiar only with traditional redundancy.

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. Figure 2(b) graphically depicts the progressive redundancy algorithm.

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 a bit more complex to compute than that of traditional redundancy. It 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 ($\frac{k+1}{2}$) do not fail, exactly the same as with traditional redundancy:

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

Figure 3 shows that for a given cost factor, progressive redundancy (labeled “PR”) always achieves a higher system reliability than traditional redundancy.

3.3. Iterative Redundancy

Distributed computational architectures typically execute job asynchronously and thus have (1) access to runtime information about system reliability and (2) the ability to alter task deployment decisions based on that information. We leverage this observation to develop *iterative redundancy* by improving on progressive redundancy.

Iterative redundancy distributes the minimum number of jobs required to achieve a desired confidence level in the result, assuming that all the jobs result 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 because of the chance that the disagreeing results are correct. In other words, the apparent risk of failure of the task is increased. The algorithm then reevaluates the situation and distributes the minimum number of additional jobs that would achieve the desired level confidence. This process is repeated until the agreeing results sufficiently outnumber the disagreeing results to reach the confidence threshold. Figure 2(c) graphically depicts the progressive redundancy algorithm.

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 sure of the result's correctness with probability \mathbb{R} . 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 four jobs and they all return the same result, there is a $\frac{0.7^4}{0.7^4+0.3^4} > 0.97$ chance that the result is correct. Four is the minimum number of jobs that can achieve the confidence threshold in this example, so the task server distributes four jobs. If all four jobs return the same result, the task is finished. However, if some jobs return a disagreeing result, 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 three 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.

Intuitively, progressive redundancy is guaranteed to distribute the fewest jobs needed to achieve the consensus. Iterative redundancy is guaranteed to distribute the fewest jobs needed to achieve a desired system reliability. Thus far, in our description of iterative redundancy, we have

avoided specifying how the technique determines this minimum number of jobs. Employing the basic intuition behind the algorithm, described above, we originally employed a relatively complex probability computation to determine the number of jobs to distribute. However, during experimentation with the iterative redundancy algorithm, we discovered a much simpler implementation that achieves the identical behavior and benefits. We first describe the complex algorithm and then the simplified algorithm.

Complex algorithm: Suppose that, of $a + b$ jobs, a return one result with probability r , and b return another result with probability $1 - r$. Consider the confidence, denoted $q(r, a, b)$, that the a jobs reported the correct result. That confidence is the probability that a jobs are right and b jobs are wrong, divided by the probability that a jobs are right and b jobs are wrong plus the probability that b jobs are right and a jobs are wrong. So $q(r, a, b) = \frac{r^a(1-r)^b}{r^a(1-r)^b + (1-r)^a r^b}$. We can use this formula to determine, given some number b of jobs that have reported a result we believe to be wrong (i.e., a result that is in the minority), how many jobs must report the result we believe to be right (i.e., a result that is in the majority) for us to be \mathbb{R} confident in the majority result. We denote that number $d(r, \mathbb{R}, b)$. Thus, $d(r, \mathbb{R}, b)$ is the minimum a such that $q(r, a, b) \geq \mathbb{R}$. We can determine $d(r, \mathbb{R}, b)$ by testing consecutive a values, or employing a more sophisticated and faster method (e.g., Newton's method [23]).

Simplifying insight: While investigating iterative redundancy, we observed that whenever a task completed, the difference between the number of majority and minority results was constant. For example, if the algorithm first sought 6 unanimously agreeing results, but got 4 agreeing and 2 disagreeing results, the algorithm would distribute 4 additional jobs (in an effort to produce an 8-to-2 majority) to achieve the desired reliability. This phenomenon arises from the somewhat counterintuitive fact that, for all j , $q(r, a, b) = q(r, a + j, b + j)$. For example, 6 agreeing results and 0 disagreeing results instills the same confidence as 106 agreeing results and 100 disagreeing results. The key to properly understanding the reasoning is that the probability of a 106-to-100 decision split occurring may be low, but once the system is faced with a 106 to 100 decision split, it is irrelevant how unlikely such a situation was to happen in the first place; given that this unlikely situation has occurred, the relevant quantity is how likely the 106 jobs are to have been correct.

Simple algorithm: Using this insight, we can greatly simplify the iterative redundancy algorithm. We only need to determine $d(r, \mathbb{R}, 0)$ once and set that quantity to be the required minimum difference d between the number of jobs reporting the majority result and the number reporting the

```

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

```

Figure 4. The iterative redundancy algorithm.

minority result. For example, if $d(r, \mathbb{R}, 0) = 6$, the algorithm iterates, automatically distributing jobs until 6 more jobs have reported one result than the other. Even further, a user may specify the desired reliability improvement in terms of the d number, and then neither the user nor our technique need know the average reliability r of nodes in the node pool. This situation is parallel to the progressive and traditional redundancy techniques, in which the user specified a parameter k . Figure 4 specifies the entire iterative redundancy algorithm in pseudocode. Despite being much simpler than and appearing to be quite different from the original algorithm, this simplified algorithm deploys the same number of redundant jobs in every situation and accomplishes the exact same efficiency.

Analysis: For iterative redundancy with d as defined above, we use $\mathbb{R}_{IR}^d(r)$ and $\mathbb{C}_{IR}^d(r)$ 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,

$$\mathbb{C}_{IR}^d(r) = \sum_{b=0}^{\infty} (d + 2b) Pr \left[\begin{array}{l} d + 2b \text{ jobs produce} \\ d + b \text{ identical results} \end{array} \right]. \quad (5)$$

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,

$$\mathbb{R}_{IR}^d(r) = q(r, 0, d) = \frac{r^d}{r^d + (1 - r)^d}. \quad (6)$$

Figure 3 shows that for a given cost factor, iterative redundancy (labeled “IR” in Figure 3) always achieves a higher system reliability than both traditional and progressive redundancy.

4. Evaluation

In this section, we analyze the costs and benefits of the three redundancy techniques. In addition to some formal arguments based on Equations (1) through (6), each of these analyses will include data from a discrete event simulation of a distributed computational architecture and a deployment of the BOINC volunteer computing system [3, 5] on the distributed PlanetLab platform [22].

We will first, in Section 4.1, describe our evaluation platforms for the simulation and BOINC deployments. We will then, in Sections 4.2 and 4.3, evaluate the *efficiency* and *self-adaptation* of the techniques, respectively.

4.1. Evaluation Platforms

We used two off-the-shelf platforms to evaluate the redundancy techniques: XDEVS [12] and BOINC [5].

XDEVS Simulation Environment

The XDEVS simulation framework [12] 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 the XDEVS simulation 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 incorporate domain-specific algorithms and constraints, ensures that the aspects of our system model enumerated in Section 2 are enforced.

Using XDEVS for empirical evaluation 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 given in this section 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, as in Section 4.3).

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.

BOINC Deployment

Our second empirical evaluation utilized the BOINC volunteer computing system [3, 5]. BOINC is a popular distributed computation architecture currently deployed on over a million machines. Examples of BOINC applications include SETI@home, LHC@home, Folding@home, Malariaccontrol.net, and Climateprediction.net. The BOINC server software [5] 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 problems into individual tasks that test whether particular Boolean assignments satisfy a Boolean formula, and (2) modified the job-assignment and result-validation procedures to employ iterative and progressive redundancy.

We deployed our BOINC system on a 200-node subset of PlanetLab [22]. The PlanetLab testbed consists of roughly 1,000 machines of varying speeds and resources, distributed at 500 locations around the world.

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 given in this section 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% of the time,
- 2: PlanetLab nodes becoming unresponsive, and
- 3: all other unanticipated failures that PlanetLab nodes might experience.

Each execution recorded the time 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, and the number of tasks that achieved a correct result.

4.2. Efficiency

In Section 3, we described our expectations for the performance of the redundancy techniques. In this section, we

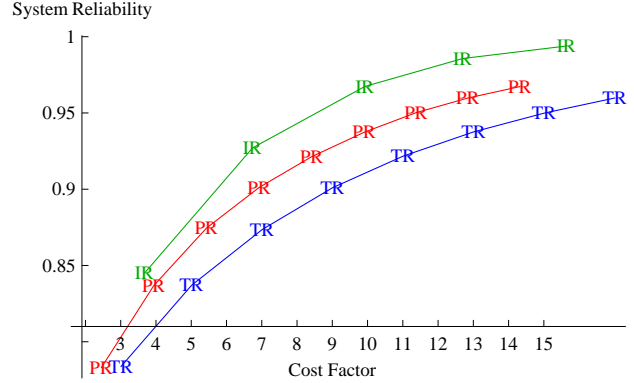


Figure 5. Experimental results from the XDEVS simulations, shown here for $r = 0.7$.

present empirical data from, first, simulated systems executed in XDEVS and, then, from BOINC systems deployed on PlanetLab to confirm our theoretical predictions.

Figure 5 shows our empirical data from the XDEVS simulations supporting the claim that iterative redundancy outperforms traditional and progressive redundancy in the number of jobs and total time to execute the computation. This data (for $r = 0.7$) closely agrees with our analytical predictions from Section 3.

The exact cost factor improvement of iterative redundancy depends on r . Figure 6 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, 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

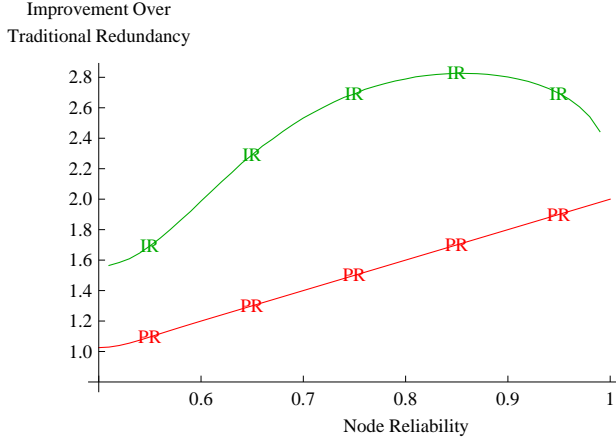


Figure 6. The ratio improvement in cost factor for progressive (PR) and iterative (IR) redundancy over traditional redundancy.

experiments to test this hypothesis.

In our next set of experiments, we deployed the redundancy techniques on a BOINC system running on Planet-Lab. 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 represented typical real-world deployments. Figure 7 depicts the system reliability as a function of the cost factor of each technique. These data points are averages of multiple executions. 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 Figure 7 allowed us to estimate the reliability of PlanetLab nodes. Each technique’s executions regularly reported costs and system reliabilities consistent with $0.64 < r < 0.67$. Our seeded faults lowered r to 0.7 and the naturally occurring PlanetLab faults were responsible for the difference. The consistency of the derived node reliabilities, both among multiple trials with different parameters and across all three techniques, provides strong evidence for the validity of these experiments.

4.3. Self-Adaptation

Iterative redundancy adapts to a changing environment in two ways: (1) by 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) by injecting extra redundancy into “unlucky” situations with disproportionately many failures. To achieve this, iterative redundancy does not bound the maximum number of

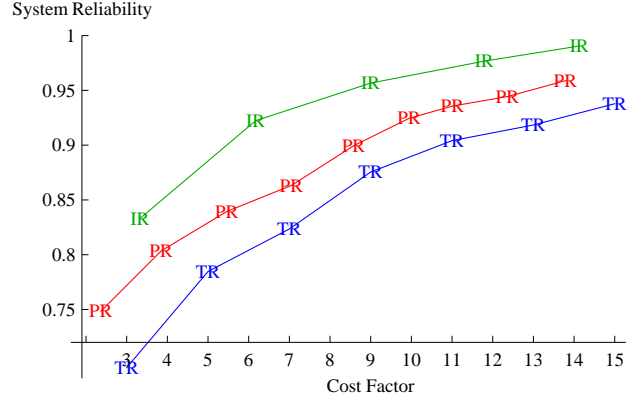


Figure 7. BOINC deployment experimental results comparing the efficiency of redundancy techniques.

jobs deployed per task; however, on average, that number approaches the cost factor.

To illustrate the first quality, we conducted a number of experiments in which the reliability of the nodes varied over time. Figure 8 shows three executions of a single system over time using each of the three techniques. Node reliability (top), varies between 0.75 and 0.95, and is the same for all three 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 resulting in a correct result drops significantly. Progressive redundancy allows the jobs per task to vary within a predefined range to adjust to changes in 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 number of tasks that result in a correct result fairly constant.

These graphs also illustrate how iterative redundancy outperforms progressive redundancy in terms of cost factor. When node reliability peaks, progressive redundancy “maxes out” system reliability and produces 100% correct results, but it cannot reduce the jobs per task below $\frac{k+1}{2}$. At these times, 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.

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. We will discuss, in Section 5, scenarios that may make progressive redundancy

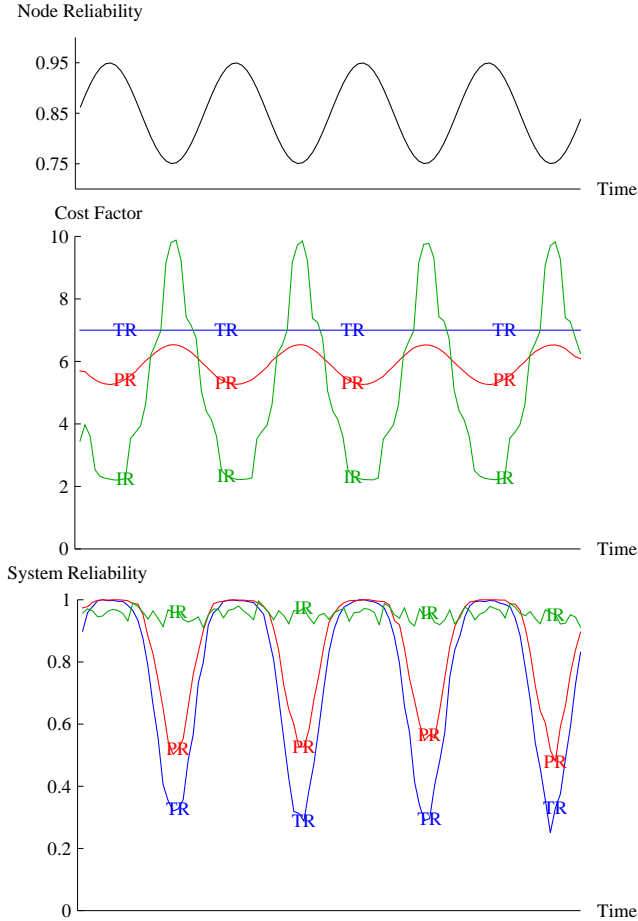


Figure 8. Traditional, progressive, and iterative redundancy techniques react differently to the changing node reliability r .

preferable to iterative redundancy.

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. Figure 9 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 disagreeing result. For traditional redundancy, the amount of redundancy is constant regardless of how many nodes disagree with the majority. When progressive redundancy is used, the total number of jobs distributed is equal to the number of disagreeing nodes plus $\frac{k+1}{2}$ (the number of nodes in the majority). Finally, the iterative redundancy technique distributes a total num-

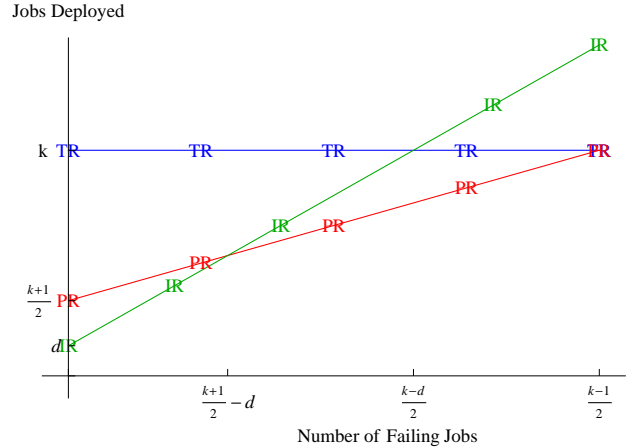


Figure 9. Iterative redundancy (IR) injects more redundancy than progressive (PR) and traditional (TR) redundancy into the cases with the most failing jobs.

ber of jobs equal to twice the number of disagreeing nodes plus d . Therefore, iterative redundancy provides an adaptive mechanism for applying additional redundancy in situations with some failing jobs. These same mechanisms allow iterative redundancy to automatically keep the confidence level across tasks relatively constant.

5. Discussion

In this section, we discuss how iterative redundancy responds to poor estimates of the reliability of nodes, the possible shortcomings of the technique, and the effect of relaxing of our earlier assumptions.

5.1. Predicting Node Reliability

We set out to design iterative redundancy to achieve a desired system reliability \mathbb{R} , given a particular reliability of the nodes. In some sense, we expected that the algorithm will need to know each job’s reliability r in order to correctly approximate how much redundancy to inject. As we already described in Section 3.3, during our exploration of iterative redundancy we discovered that a much simpler algorithm that does not use r or \mathbb{R} performed exactly the same actions as our original goal algorithm. The user only needed to specify how much improvement was needed (or how high a cost in execution time she was willing to pay) and the algorithm used those resources to achieve the highest possible system reliability.

In addition to our theoretical analysis, we wanted to verify our discovery experimentally. To that end, we used our

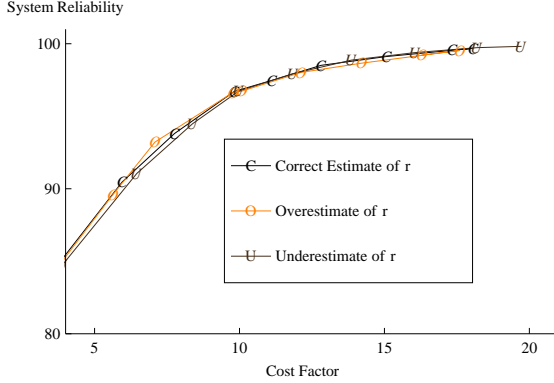


Figure 10. The iterative redundancy algorithm is robust to poor estimates of r .

XDEVS simulation engine to simulate multiple sets of three systems attempting to achieve a given system reliability \mathbb{R} : the first system in each set overestimated the reliability r , the second estimated r correctly, and the third underestimated r . Figure 10 shows that, for a fixed cost factor, iterative redundancy produced the same system reliability regardless of the estimated r , supporting our theoretical conclusions.

5.2. Possible Shortcomings

We have concentrated on minimizing the number of jobs needed to execute in order to complete computations. 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 response time for a particular task. In the realm of computational grids with a large number of independent tasks, the increased response time does not present a problem because the nodes can always execute jobs related to other tasks [3, 11]. However, some applications may pose requirements on the response time 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.

5.3. Relaxing Assumptions

We have made several assumptions, described in Section 2.3, that allowed us to more clearly explain how iterative and progressive redundancy can inject robustness into distributed computation architectures such as computational grids, volunteer computing systems, and MapReduce implementations. We had 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. In this section, we explain that redundancy can apply to distributed computation architectures deployed on networks without these assumptions, and, in some cases, can even benefit from their relaxation.

We have already described, in Section 5.1, that the user does not need to know r to use iterative redundancy. However, knowing r can help calculate the reliability of the systems employing the technique. An improved estimate of r will result in a more accurate calculation of system reliability. It is also possible to leverage information about distinct r values for every job and dependency between job failures in calculating the system reliability. Equations (1) through (6), as well as the analysis in Section 4, reflect the assumption that each job has an equal probability of failure. We made this assumption based on the fact that many distributed computation architectures (e.g., BOINC [3] and MapReduce [11]) 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. In some circumstances and for certain classes of jobs, it may be possible to estimate the distinct reliability information for different classes of nodes and jobs. Further, probabilities of node and job failure 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. In such cases, the job schedulers can use the additional information to decrease the probability of failure. 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) would become

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

The final cost and probability of failure would then depend on the probability distribution. 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. In an attempt to use node reliability knowledge to improve efficiency, BOINC has recently added adaptive replication, the ability to prevent replication of a task if a trusted node returns its result.

The assumption that the result of every task is a single bit has simplified our analysis thus far, but it actually turns out to be the worst-case scenario. Compare two types of tasks, for example: 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 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.

6. Related Work

In this section, we (1) discuss several distributed computation architectures to which iterative redundancy applies and (2) contrast iterative redundancy with existing fault-tolerance approaches.

6.1. Distributed Computation

We have already discussed the BOINC distributed computation architecture [3, 5] in Sections 4.1 and 5.3, and do not repeat ourselves here. MapReduce [11] is another well-known distributed computation architecture, in which 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 [11]. As its core, that MapReduce infrastructure does not use redundancy; however, Hadoop [16], a popular implementation of MapReduce, leverages traditional redundancy in its file system to reliably store data. While Hadoop can redeploy failed jobs, its support for redundancy is rudimentary at best.

6.2. Fault-Tolerance

Reliability is a well-studied field in software systems and some approaches have emerged to leverage redundancy to improve a system’s reliability. As mentioned above, we based our progressive redundancy on a self-configuring optimistic programming technique [7, 6] aimed at component-based systems. Such systems allow for asynchronous job scheduling; however, they focus on minimizing response time and typically must allocate finite resources to every task. Distributed computation architectures can relax these limits, which allows iterative redundancy to deploy jobs without (1) a priori knowledge of node reliability and (2) a bound on the total number of necessary jobs.

Most redundancy implementations fall into one of two categories: primary backup [8] and active replication [13]. Primary backup uses multiple servers to improve the reliability of a service. One server is designated as the primary server 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 [8]. Iterative redundancy complements primary backup by specifying, at runtime, how many backups should exist to guarantee a particular level or reliability.

Active replication removes the centralized control of primary backup and minimizes losses that occur when a subset of the replicas fail. Active replication incurs a high cost associated with keeping all the replicas synchronized [13]. 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 mechanisms in distributed systems, iterative redundancy improves the efficiency of those mechanisms.

Hwang [17] proposed a method for injecting “smarter” fault tolerance into grids that suggests the possibility of

handling a wide variety of faults within distributed systems. This work provides a service to detect crash failures (and an extension to allow the system designer to specify other failures and how to detect them) and a failure-handling framework that enforces designer-defined policies [17]. Traditional checkpoint techniques can also be applied to distributed computation architectures 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 [24].

Finally, autonomous agents capable of detecting failing components and initiating on-demand replication allow autonomous fault tolerance, although the developer has to implement fault-specific detection mechanisms into these agents [10].

7. Contributions

We presented a novel technique, iterative redundancy, that improves on existing reliability techniques by leveraging runtime information. We identified several types of systems to which iterative redundancy applies and concentrated in this paper on distributed computation architectures. Iterative redundancy is more efficient than existing methods in its use of resources. It is adaptive because (1) it increases redundancy on-the-fly when component reliability drops and decreases redundancy when component reliability rises, and (2) it injects redundancy where it is beneficial and eliminates it where it is unnecessary.

In addition to a rigorous theoretical analysis of iterative redundancy, we presented an empirical evaluation based on two deployments: the XDEVS discrete event simulator and the BOINC volunteer computing system. In the process of conducting and evaluating this work, a spectrum of possible redundancy techniques has begun to emerge, with the two extreme points on that spectrum being traditional redundancy and iterative redundancy. We have identified a number of interesting research questions, which frame our current and future work.

References

- [1] M. Abd-El-Malek et al. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, pages 59–74, 2005.
- [2] M. K. Aguilera et al. Consensus with Byzantine failures and little system synchrony. In *DSN*, pages 147–155, 2006.
- [3] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *GRID*, pages 4–10, 2004.
- [4] J. Andrade et al. Using grid technology for computationally intensive applied bioinformatics analyses. In *Silico Biology*, 6(0046), 2006.
- [5] BOINC. The Berkeley open infrastructure for network computing. <http://boinc.berkeley.edu>, 2009.
- [6] A. Bondavalli et al. A cost-effective and flexible scheme for software fault tolerance. *J. of Computer Systems Science and Engineering*, 8(4):234–244, 1993.
- [7] A. Bondavalli et al. An adaptive approach to achieving hardware and software fault tolerance in a distributed computing environment. *J. Systems Architecture*, 47(9):763–781, 2002.
- [8] N. Budhiraja et al. The primary-backup approach. In *Distributed Systems*, pages 199–216. 2nd edition, 1993.
- [9] A. J. Chakravarti and G. Baumgartner. The organic grid: Self-organizing computation on a peer-to-peer network. In *Intl. Conf. Autonomic Computing*, pages 96–103, 2004.
- [10] A. De Luna Almeida et al. Towards autonomic fault-tolerant multi-agent systems. In *Latin American Autonomic Computing Symposium*, 2007.
- [11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation*, 2004.
- [12] G. Edwards and N. Medvidovic. A highly extensible simulation framework for domain-specific architectures. Technical Report USC-CSSE-2009-511, 2009.
- [13] P. Felber and A. Schiper. Optimistic active replication. In *ICDCS*, pages 333–341, 2001.
- [14] A. D. Friedman and P. R. Menon. *Fault Detection in Digital Circuits*. Prentice Hall, 1971.
- [15] The Globus alliance. <http://www.globus.org>, 2005.
- [16] Hadoop. <http://hadoop.apache.org>, 2009.
- [17] S. Hwang and C. Kesselman. A flexible framework for fault tolerance in the grid. *J. Grid Computing*, 1(3):251–272, 2003.
- [18] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.
- [19] T. Kimoto et al. Stock market prediction system with modular neural networks. In *Intl. Joint Conf. Neural Networks*, pages 1–6, 1990.
- [20] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Elsevier, Inc., 2007.
- [21] M. Lamanna. The LHC computing grid project at CERN. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 534(1–2):1–6, 2004.
- [22] L. Peterson et al. A blueprint for introducing disruptive technology into the Internet. *ACM SIGCOMM Computer Communication Review*, 33(1):59–64, 2003.
- [23] W. H. Press et al. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007.
- [24] S. B. Priya et al. Fault tolerance-genetic algorithm for grid task scheduling using check point. In *Intl. Conf. Grid and Cooperative Computing*, pages 676–680, 2007.
- [25] A. Setiawan et al. GridCrypt: High performance symmetric key cryptography using enterprise grids. In *Intl. Conf. Parallel and Distributed Computing: Applications and Technologies*, pages 872–877, 2004.
- [26] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.