Combining Genetic Algorithms and Simulation to Search for Failure Scenarios in System Models

Kevin Mills, Christopher Dabrowski, James Filliben and Sandy Ressler
{kmills, cdabrowski, jfilliben, sressler}@nist.gov
Information Technology Laboratory
NIST
Gaithersburg, MD, USA

***Abstract*—Large infrastructures, such as clouds, can exhibit substantial outages, sometimes caused by failure scenarios not predicted during system design. We define a method for model-based prediction of system quality characteristics. The method uses a genetic algorithm to search system simulations for parameter combinations that result in system failures, so that designers can take mitigation steps before deployment. We apply the method to study an existing infrastructure-as-a-service cloud simulator. We characterize the dynamics, quality, effectiveness and cost of genetic search, when applied to seek a known failure scenario. Further, we iterate the search to reveal previously unknown failure scenarios. We find that, when schedule permits and failure costs are high, combining genetic search with simulation proves useful for exploring and improving system designs.**

Keywords—genetic algorithms; model-based prediction; simulation methodology; system design

I. INTRODUCTION

Modern society grows increasingly dependent on large infrastructures, such as clouds, for computation and storage needs. Yet, such infrastructures are prone to substantial outages, which sometimes arise from failure scenarios that were not considered during system design [1]. Because the state-space of such systems is vast [2], discovering potential failure scenarios is quite difficult, somewhat like searching for a needle in a haystack. A collection of evolutionary computation methods [3] exists to search for optimal solutions in large spaces. Among those methods, genetic algorithms (GAs) [4] provide a flexible approach, well suited for problems where little information is available about the structure of the solution space. Here, we investigate combining a GA with simulation to search system designs for *anti-optimal* solutions, e.g., parameter combinations that yield degraded performance, such that only a small percentage of a system's users can successfully obtain needed resources. We demonstrate that this approach can help system architects to identify potential failure scenarios before system deployment.

Section II presents related work. Section III illustrates our approach; describes the GA used in our case study, including key control parameters; discusses minimum requirements for a system simulator to be controlled by a GA; and outlines an iterative process to hunt for failure scenarios. Section IV describes a case study, applying the GA to search for a known failure scenario in an existing cloud simulator. In addition to introducing the simulator and related parameters, we explain how the GA maps simulator parameters to 'chromosomes'. Further, we illustrate our deployment of GA search as a distributed application on a cluster, and we define settings used for key GA control parameters. Section V presents and discusses results from our case study, illustrating the dynamics, quality, effectiveness and costs of GA search for a known failure scenario. We also discuss previously unknown failure scenarios discovered in subsequent repetitions of GA search. Section VI gives conclusions and future work.

II. RELATED WORK

Modern information systems are increasing in complexity: growing in size and geographic scope, changing constantly as software and hardware components are added and removed, and providing shared support to users with many different applications. These traits make failure scenarios both difficult to foresee and expensive to experience; thus, researchers actively pursue prediction techniques that can be applied during system design (offline) and at run time (online).

Available research literature generally explores design-time methods for complementary purposes: (1) improving system models or, like the current paper, (2) exploring system models to identify potential failure scenarios. One main goal of improving system models is to provide better probability estimates for rare events. Better estimates can help to parameterize system models with more realistic failure distributions. Researchers investigate two main approaches: splitting [5-7] and importance sampling [8], or some combination [9]. The main goal of such techniques is variance reduction for probability estimates of rarely occurring events. The main mechanism is to steer simulations into regions that generate more samples of rare events, which would otherwise occur infrequently and, thus require long simulation times in order to generate accurate estimates.

Some researchers use system models to search for failure scenarios that might otherwise be overlooked during design. For example, Shultz and colleagues [10] used an approach similar to ours, applying a GA to seek combinations of faults that cause anomalies in control behavior within two simulators, an autonomous aircraft and a submersible. Their work differs from ours in two main ways. First, they devised domain-specific encodings to match fault-scenario languages that were unique to each simulator. Our method uses classical GA binary encoding that can be applied generally to any simulator that can be parameterized numerically. Second, they used domain-specific knowledge to modify the usual genetic operators. We

 ISBN: 978-1-61208-308-7

applied a classical GA without modifying the crossover and mutation operators. Yucesan and Jacobson [11] used simulated annealing (SA) to search for event sequences leading to failed termination conditions. They report that SA has a major drawback: requiring customization of control parameters to suit particular problems. Our approach used GA control parameters selected [12] independently of specific problems. Dabrowski and Hunt [13] used graph analysis to search for cut sets (indicating failure vulnerabilities) in Markov chain models, derived from detailed system simulators. After identifying vulnerabilities, they used perturbation analysis to determine failure thresholds and trajectories. Their method involved modeling processes as zero-order Markov chains, which are "memoryless", and thus do not capture behavioral history. Our approach directly explores detailed system simulators, including historical behaviors, allowing assessment of detailed system processes. Fainekos and colleagues [14] propose a variety of optimization techniques (e.g., GAs, ant-colony optimization, Monte Carlo simulations and cross-entropy method) to search for parameter combinations that violate invariant execution trace properties expressed using Metric Temporal Logic. Exploiting such approaches requires a specification of desired behaviors in order to look for violations. Our approach requires only a single measure of anti-fitness, which does not require a rigorous statement of desired system properties.

While design-time methods seek failure scenarios that could arise after deployment, run-time methods aim to make specific, timely predictions of impending failures in deployed systems, so that system operators can take remedial actions. As Matsuo notes [15], predicting future behavior in complex systems appears quite difficult. Yet, even moderate success can have large positive returns, which encourages researchers to continue investigating the efficacy of many approaches [16] that might predict failures during system operation. Most researchers apply offline techniques, such as machine learning [17-18], hidden Markov models [19], GAs [20], and Bayesian estimation [21-22] to learn patterns, which online systems can monitor to predict failures. Other researchers [23-24] explore monitoring techniques without an offline component.

III. METHOD

Figure 1 gives a schematic of our method, where a GA controls a population of simulators distributed on a high-performance computing cluster. First, an analyst selects a list of simulator parameters to search, defining for each a range and granularity; thus each parameter can take on a discrete set of values. The GA uses this information to construct an internal representation, or 'chromosome' map, specifying the location and number of bits representing each parameter (see Table II, for an example). Subsequently, the GA instantiates random bit strings (i.e., chromosomes) for each simulator in the population, and then transforms them to parameter files. Each simulator reads its assigned file, updates its internal parameters accordingly and runs a simulation. As each run finishes, the corresponding simulator reports a metric, which we call *anti-fitness*, defined by an analyst to represent a measure of system failure. For example, in our case study (Section IV), we define anti-fitness as the proportion of users who could not be served by a simulated cloud. After collecting anti-fitness reports from the current population, the GA uses an algorithm [12] to construct a next generation of parameter combinations, and distributes a combination to each simulator in the population. Over multiple generations, tuples are collected (for later analysis) giving parameter settings and corresponding anti-fitness from each simulation run.

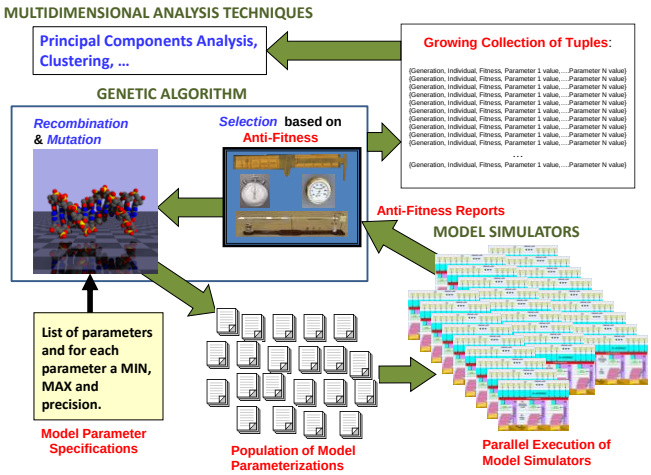

Figure 1. Schematic of a GA steering a population of simulators.

As generations progress, the GA steers the population of simulators toward parameter combinations that maximize the defined anti-fitness metric. GA steering is based solely on anti-fitness measures achieved by each set of bit strings (i.e., chromosomes). The search process is blind to the existence of parameters. The GA searches only for bit strings that achieve maximum anti-fitness. The connection between bit strings and model parameters occurs when the GA converts bit strings to parameters for input to simulators. For that reason, a GA is quite general and flexible, as we explain next.

The GA begins by generating randomly a seed population of individuals, each consisting of an appropriate length bit string, representing values for the chosen parameters of a problem to be solved. The *population size* is a control parameter of the GA. The GA next evaluates the anti-fitness of each individual. Over time, the GA evaluates the anti-fitness of many populations, where each population is called a generation. The population for generation n+1 is created through some transformation of individuals composing generation n. The GA terminates after a specified number of generations, unless terminated manually.

After evaluating a generation, the GA determines whether the population should be rebooted, which involves randomly regenerating all or part of the next generation. Rebooting can increase the GA's exploration of the search space. The GA includes a control parameter, *reboot proportion*, which defines the percentage of generations to complete before a reboot.

When selecting individuals for generation n+1, those with highest anti-fitness (i.e., the elite) from generation n can be included unchanged. The GA has a control parameter, *elite selection percentage*, which defines how many individuals from generation n will be placed unaltered into generation n+1. Such elite individuals can be placed into a population whether

 ISBN: 978-1-61208-308-7

or not the remaining individuals will be generated randomly or by transformation. When the population for generation *n*+1 is created by transformation, the procedure involves selecting a pool of candidate individuals from generation *n*, and then applying two genetic operators, crossover and mutation, which mimic reproduction in biological populations.

The GA includes a control parameter, *selection method*, which defines the algorithm used to select individuals from generation *n* for inclusion into the candidate pool, where the more anti-fit individuals from generation *n* may be included multiple times, while some of the least anti-fit individuals may be excluded. Given a pair of individuals, chosen randomly from the candidate pool, a GA control parameter, *number of crossover points*, determines where bits will be swapped between them. The specified number of crossover locations is chosen randomly, uniformly distributed within the length of bit strings comprising chromosomes, and the bits in each individual are swapped at those points and the two transformed individuals are placed into the population of recombined individuals. This continues until a sufficient number of (possibly) transformed individuals are selected to fill a population.

Subsequently, the GA iterates over each bit representing each recombined individual, while deciding whether or not the bits should be inverted. A GA control parameter, *mutation rate*, specifies the probability that any given bit will be flipped. After mutation, the GA inserts the individual into the population for generation *n*+1. Subsequently, the anti-fitness of each individual is evaluated and the GA iterates through generation *n*+1, creating the population of individuals for generation *n*+2, and so on until termination.

For a GA control, a simulation model must be able to initialize its parameters from external inputs and must be capable of executing in a loop, as simulations finish and new inputs arrive. Most simulators have such capabilities, but must be modified to asynchronously await inputs generated by a GA and to report anti-fitness once the simulation completes. We made these modifications via a signal file shared between each simulator instance and the GA. As discussed in Section V.A, more substantive simulator modifications were also necessary.

Once a complete search is finished, several potential failure scenarios may be revealed (see Section V). The analyst is then free to resolve those failures and repeat the GA search for additional scenarios. This iterative process can continue until no more failure scenarios appear or available time is exhausted.

IV. CLOUD SYSTEM CASE STUDY

We evaluated GA search while seeking a previously known failure scenario [25] in Koala, an existing infrastructure-as-a-service (IaaS) cloud simulator. The simulator architecture is shown in Figure 2. A full description of Koala can be found elsewhere [26-27]. Here, we give only a summary.

Koala simulates five layers: (1) demand from users, each requesting a collection of virtual machines (VMs), (2) a supply of physical nodes on which VMs can be placed, (3) a resource allocation layer, consisting of a cloud controller and cluster controllers that cooperate to determine a cluster on which to place VM collections, (4) an Internet/Intranet layer providing communication among simulated nodes, and (5) a VM behavior layer that models variations in resource usage over time. Available VM types and physical platforms are modeled after the Amazon EC2 Cloud, while the three-tier cloud architecture (cloud, cluster and node controllers) is modeled after a public domain version (1.6) of Eucalyptus. (Mention of commercial products or organizations in this paper does not imply endorsement by NIST.)

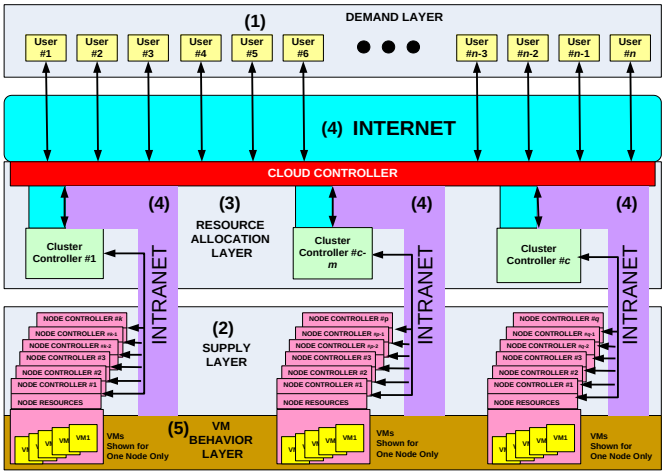

Figure 2. Schematic of Koala IaaS cloud simulator.

All nodes are placed geographically in a coordinate system of sites, where the cloud elements may be placed on one or multiple sites. User sites are selected randomly upon each user's arrival; user types are also assigned randomly. User type determines the quantity and mix of VMs the user will request, which can include a minimum number required to start an application and a maximum that can be exploited should sufficient resources be available. A user requests VMs, and the cloud controller can honor the request fully (maximum requested VMs) or partially (at least minimum requested VMs). If insufficient resources exist, the cloud responds with NERA (not enough resources available). Upon receiving a NERA response, a user may retry intermittently during a day, and if VMs cannot be obtained, then retire for the evening and return the next day to try again. After passing too many days without obtaining the needed VMs, the user gives up and leaves the system, only to be regenerated as a new user.

Upon successfully obtaining VMs, a user selects a holding time, during which VMs may be added or terminated. Of course, VMs may also fail within the cloud, so a user will attempt to maintain a required minimum number of VMs by requesting additional VMs as needed. When holding time expires, a user requests termination of all VMs and reenters the system as a new user.

The cloud controller handles all user requests, checking with subordinate cluster controllers to find available space for a collection of VMs, which are mapped to a single cluster to localize inter-VM communication. The cloud controller uses one of several algorithms [27] to select a specific cluster. Cluster controllers monitor the state of subordinate nodes, and use one of several algorithms [27] to select specific nodes for

 ISBN: 978-1-61208-308-7

placement (or relocation) of individual VMs. Under guidance, from a simulated administrator, the cloud controller can add and remove clusters from the cloud, and cluster controllers can add and remove nodes from clusters. The administrator can also terminate VMs that the cluster controller is unable to stop.

Table I categorizes 129 Koala parameters over which we conducted a GA search. More than half the parameters define element behaviors, most by the user and cloud controller, while 22 describe structural elements, half related to the network. The model also simulates failures that could occur in the network and among the physical platforms and components. A smaller set of parameters can inject behavioral and structural asymmetries, such as changing user demand profiles over time and allowing clouds to be constructed as a combination of large and small clusters, rather than of same-sized clusters.

TABLE I. SUMMARY OF KOALA PARAMETERS TO SEARCH OVER.

Model Element	Parameter Category				
	Behavior	Structure	Asymmetry	Failure	Total
User	28	2	4	0	34
Cloud Controller	21	4	5	0	30
Cluster Controllers	11	5	3	0	19
Nodes	6	0	0	14	20
Intra-Net/Inter-Net	4	11	2	9	26
Totals	70	22	14	23	129

Among the 129 parameters, we included four Booleans to turn on/off behaviors that control orphan VMs, a potential problem uncovered in earlier experiments with Koala [27], where lost messages could leave users and the cloud controller unaware that VMs had been allocated, leading to retries, reallocations, and ultimately to saturation of cloud resources. Most orphan VMs arise during initial allocation, but some are caused by failed terminations. Additionally, orphan VMs may occur as collections of VMs are relocated before a cluster is shut down. Logic was included in Koala to detect and remove orphans in all three classes, and an administrator was also simulated to allow residual orphans to be stopped manually.

When a cloud is saturated with VM orphans, users are unable to obtain requested VMs and eventually give up after exhausting their patience. For our GA search, we defined anti-fitness as the ratio of users who give up to the total number of arriving users. The larger the ratio, the more users were turned away, and the lower the cloud's revenue.

To guide the GA search, we defined a range and precision for each Koala parameter (see Table II for an elided list), yielding a mean of about six values per parameter, and thus a search space of approximately 10^{100} parameter combinations. Using our description, the GA computed the number of values (and bits) needed to encode a Koala parameter combination (as a binary string), and then randomly placed the binary encoding for each parameter into a 334-bit chromosome, which served as the internal form used by the GA to represent Koala parameters. The GA also provided routines to generate Koala parameter values from binary encodings given in chromosome form.

We deployed a distributed population of 200 Koala simulators on a high-performance cluster, under GA control (see Figure 3) via signal files in a shared, network file store. Each simulator, allocated to one core, waits for the GA to signal that a parameter file exists and then runs a simulation, reports the resulting anti-fitness value, and awaits the next signal. The GA generates parameter files for each simulator and periodically checks progress and collects anti-fitness reports as runs complete. Once all runs in a generation finish, the GA uses the algorithm described in Section III to create the next generation of parameter files, and so on until completing a specified number of generations.

TABLE II. MAPPING OF KOALA PARAMETERS TO CHROMOSOMES.

	Koala Parameter Space (Size = 10^{100})			Genetic Algorithm Computed Chromosome Map (Size = 2^{334})			
PARAMETER	MIN	MAX	PRECISION	#VALUES	LOW_BIT	HIGH_BIT	#BITS
P_CreateOrphanControlOn	0	1	1	2	36	36	1
P_TerminationOrphanControlOn	0	1	1	2	58	58	1
P_RelocationOrphanControlOn	0	1	1	2	11	11	1
P_AdministratorActive	0	1	1	2	330	330	1
P_clusterAllocationAlgorithm	0	5	1	6	31	33	3
P_describeResourcesInterval	600	3600	600	6	81	83	3
P_nodeResponseTimeout	30	90	30	3	210	211	2
P_TerminatedInstancesBackOffThreshold	3	6	1	4	56	57	2
P_TerminationBackOffInterval	180	360	60	4	88	89	2
P_TerminationRetryPeriod	600	1200	300	3	316	317	2
P_StaleShadowAllocationPurgeInterval	600	3600	600	6	242	244	3
P_cloudAllocationCriteria	0	3	1	4	321	322	2
P_clusterShadowPurgeLimit	1	21	5	5	290	292	3
P_instancePurgeDelay	180	600	60	8	98	100	3
P_clusterEvaluationResponseTimeout	60	120	30	3	14	15	2
P_MaxPendingRequests	1	10	1	10	72	75	4
P_CloudTerminatedInstancesBackOffThreshold	3	6	1	4	169	170	2
P_CloudTerminationBackOffInterval	180	360	60	4	40	41	2
P_CloudTerminationRetryPeriod	3600	10800	1800	5	297	299	3
P_ClusterShutdownGracePeriod	86400	2.59E+05	43200	5	147	149	3
●	●	●	●	●	●	●	●
P_RequestEvaluatorTimeoutWaitProportion	0.1	0.4	0.1	4	145	146	2
P_RequestEvaluatorClusterMinimumResponse	0.6	0.9	0.1	3	269	270	2
P_MaxRelocationDuratonProportion	0.65	0.95	0.1	4	90	91	2
P_MaximumRelocateDescribeRetries	4	16	2	7	254	256	3
P_AverageCloudAdministratorAttentionLatency	28800	86400	14400	5	308	310	3
P_AverageCloudAdministratorShutdownDelay	300	900	300	3	45	46	2
P_avgTimeToClusterCommunicationCut	2.88E+06	2.88E+07	2.88E+06	10	217	220	4

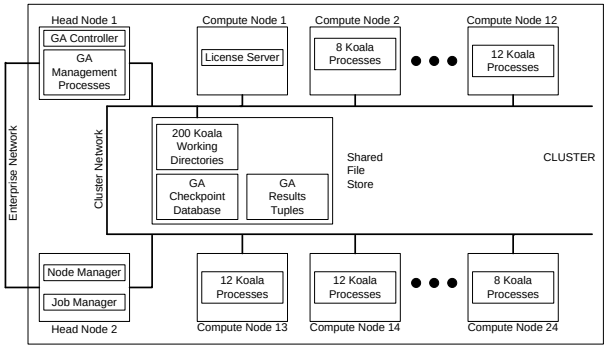

Figure 3. Schematic of simulators deployed on a cluster under GA control.

TABLE III. SETTINGS FOR KEY GA CONTROL PARAMETERS.

Generations	500
Population Size	200 Individuals
Elite Per Generation	16 Individuals
Reboot After	200 Generations
Selection Method	Stochastic Uniform Sampling
# Crossover Points	3
Mutation Rate	0.001 ≤ Adaptive ≤ 0.01

Table III shows settings we assigned for key GA control parameters. Mutation rate is controlled by an adaptive algorithm that increases mutation probability (and variance among parameter combinations) as the range of population fitness narrows and lowers probability upon divergence.

 ISBN: 978-1-61208-308-7

V. RESULTS AND DISCUSSION

To assess dynamics, quality, effectiveness, and cost of GA search, we steered a population of 200 Koala simulators over 500 generations, expecting the previously known VM orphan problem to be revealed in cases where Koala's orphan-control logic was disabled. Figure 4 shows three plots, where the *y*-axis gives (a) average, (b) standard deviation and (c) maximum anti-fitness vs. time (increasing generations). Mean anti-fitness starts low (around 0.2) for randomly generated parameter combinations, and peaks (at 0.79) within 11 generations, before falling to around 0.65 (2/3 of users not served) until generation 201 (also 401), when the GA randomizes parameters for the 184 non-elite individuals. After these reboots, mean anti-fitness rises quickly to over 0.7 and then falls back to around 0.65. Our shared file store suffered a disk crash, which required us to restart generation 311, using checkpoint information we save during the GA search. The restart caused a spike in mean anti-fitness, before settling back to around 0.65.

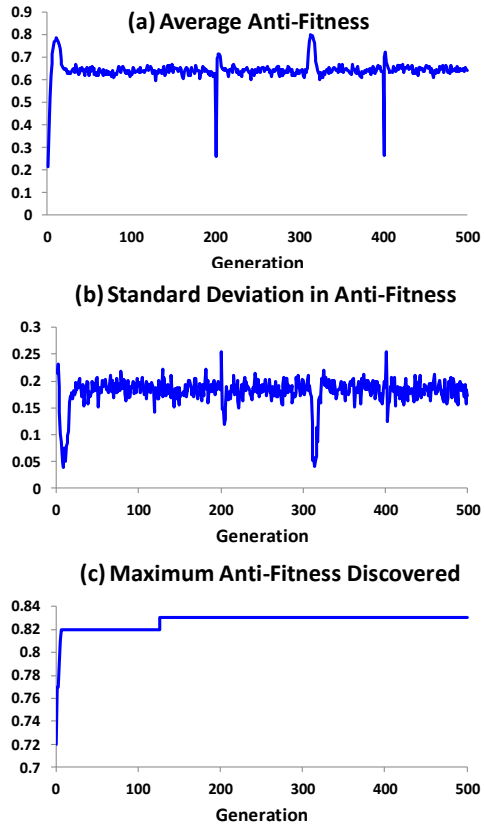

Figure 4. GA search dynamics in anti-fitness (*y*-axis)–(a) average, (b) standard deviation and (c) maximum–over 500 generations (*x*-axis).

The plot of standard deviation in anti-fitness inversely mirrors the average, i.e., high averages indicate low variance. As described previously, changes in the anti-fitness variance in a population stimulate automatic adjustments in mutation rate. The plot of maximum fitness shows that by generation 7 the GA had discovered scenarios where 82% of users could not be served, and by generation 127 the GA found scenarios where the proportion of non-served users increased to 83%. These results suggest that, for the Koala simulator, GA search could uncover failure scenarios within 100-200 generations.

Figure 5 gives a frequency distribution of anti-fitness values obtained for the (200 individuals × 500 generations =) 10^5 scenarios explored by the GA, which represent only a tiny fraction of the 10^{100} possible Koala scenarios. The histogram reveals that 84% of the scenarios explored by the GA yielded anti-fitness ≥ 0.50, despite the likelihood that most of the Koala search space consists of scenarios with low anti-fitness, as shown by the fact that randomly generated scenarios yielded mean anti-fitness of 0.2. Further, only 8.12% of the scenarios explored by the GA were duplicates, which is only slightly larger than the 8% elite individuals carried unchanged from generation to generation. These results indicate that the GA search explored predominantly non-duplicative scenarios with high anti-fitness.

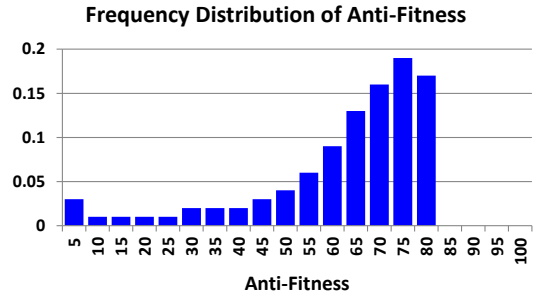

Figure 5. Histogram of anti-fitness values for all 10^5 parameter combinations.

Various analysis methods, such as feature extraction and clustering, may yield insights into failure causes. Here, we use differential probability analysis; comparing the estimated probability of each parameter-value (PV) pair appearing in scenarios with high anti-fitness against estimated probability of the same pair appearing in scenarios with low anti-fitness. We postpone, for future work, using additional analysis methods.

Let C be the set of collected tuples (recall Figure 1), each containing a vector of PV pairs and a corresponding anti-fitness value, f. We segmented C into high-pass (H) and low-pass (L) subsets: $H = \{x \in C \mid f_x > 0.70\}$ and $L = \{x \in C \mid f_x < 0.15\}$. For each PV in the high-pass subset, we estimated the probability of occurrence, $P(PV_i \mid f > 0.70)$, using the ratio $|PV_i \in H| / |H|$, representing the count of PV_i in the high-pass subset divided by the subset cardinality. We computed a similar estimate, $P(PV_i \mid f < 0.15)$, for each PV in the low-pass subset. Subsequently, we took the difference between the two estimates, $D = P(PV_i \mid f > 0.70) - P(PV_i \mid f < 0.15)$. A large positive difference suggests that a PV pair contributes to a failure scenario, while a large negative difference suggests that a PV pair contributes to desirable system behavior. Figure 6 plots D for 684 PV pairs, sorted by decreasing D, found in our GA search for a known failure scenario. We label significant outliers.

Figure 6 illustrates that most PV pairs exert little influence on failure or success scenarios, appearing about as often in both the H and L subsets. Six PV pairs appear to drive failure scenarios, and one PV pair shows most influence on success. The largest positive difference (0.58) occurs in the absence of logic to control orphans during initial VM allocation, while the largest negative difference (-0.58) occurs when that logic is present. In effect, this is the known failure scenario that we were expecting the GA to find. The second highest positive

 ISBN: 978-1-61208-308-7

difference (0.42) occurs when users select random request timeouts with an average of 30 s. By not waiting long enough for responses from the cloud, users create *virtual* message losses, because the receiving process has terminated before a response arrives. Without orphan-control procedures running, lost messages lead to a buildup of orphan VMs, leaving few resources available to serve users. This combined effect of short user timeouts and lack of orphan-control procedures was previously unknown to us. From these results, a designer might deduce that orphan-control procedures are needed, and that the cloud must find some means to ensure clients wait long-enough for the cloud to respond to requests.

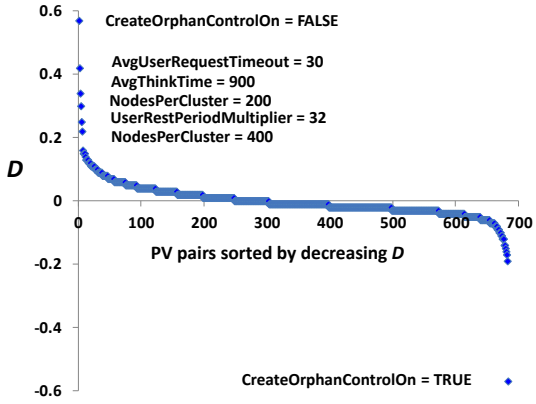

Figure 6. *D* (*y*-axis) for 684 sorted PV pairs (*x*-axis) for first GA search–outlier PV pairs labeled.

From the data in Figure 6, we were also able to identify two other potential failure scenarios: (1) cloud overload and (2) impatient users. When average cluster sizes were small (either 200 or 400 nodes), the cloud had insufficient resources to serve user demand. When the average user rest period (think time × rest period multiplier) was 8 hours, users tended to retry more frequently, and thus to give up in a shorter overall time.

A. *Costs of GA Search*

GA search for failure scenarios incurred costs of two types. First, substantial programming effort was required prior to the search. Second, GA search of simulation models can incur significant latency. We discuss each type of cost in turn.

Although the Koala simulator had been used for several years and executed robustly under diverse parameter settings, generating an initial population of random parameter combinations led to many crashes due to execution paths that were not previously encountered. Finding and fixing these software errors required significant effort. Further, the Koala simulator typically executes for a specified simulated time. The associated wall-clock time can vary widely depending upon the specific parameter settings used. To ensure deterministic search time, we modified Koala to terminate a simulation when either simulated time expired or a predetermined allocation of wall-clock time was reached. Though this was a relatively simple change, the Koala simulator had not been coded with the expectation that simulations could terminate from any given dynamic system state. Subsequently, abrupt terminations revealed many more simulator crashes, which had to be diagnosed and fixed.

Even after the Koala simulator was made sufficiently robust, numerous issues arose regarding the use of a cluster for executing simulator populations. Upon node failure, the cluster would restart simulators on some other available node. When the entire cluster failed and restarted, race conditions ensued among various components. Diagnosing the state of the entire simulator population proved difficult when using only available cluster and node management tools. To resolve such issues, it required significant effort to create a robust management system to control the population of simulators.

Executing a GA search can require substantial latency because all simulators in a given generation must complete before a next generation can be constructed. For our experiments, we limited each simulation to use no more than 90 minutes, which meant that we could complete 500 generations in 30 days. Our results showed, however, that for the Koala model we could generate failure scenarios within 100-200 generations. For that reason, we limited subsequent GA search iterations to about 200 generations, which typically complete within 14-16 days. These latency computations assume sufficient processors (one per simulator) are available for use over the entire search. If fewer processors are available, the search can take longer, though often shorter simulations can complete on a sequentially shared processor, while longer simulations execute on other processors. We completed iterative GA searches of 500, 205, 209, and 205 generations, which required a total of 74 days. These latencies suggest that GA search should be pursued only for systems with sufficient development time, and where failure scenarios have high cost.

B. *Additional Iterations of GA Search*

We conducted a second GA search; this time ensuring that orphan-control procedures and the cloud administrator were always active. Our goal was to evaluate the ability of GA search to find additional failure scenarios. We executed only 205 generations. Figure 7 plots estimated probability differences for the 677 PV pairs found by the GA.

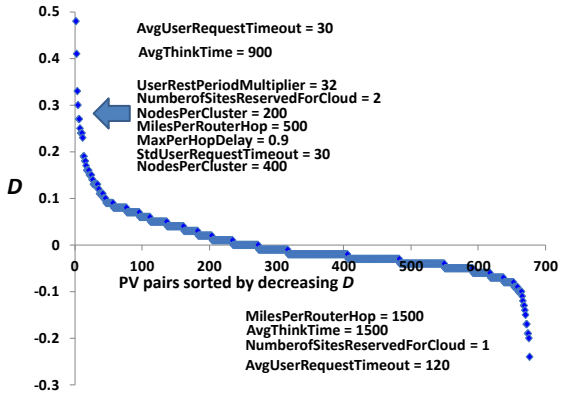

Figure 7. *D* (*y*-axis) for 677 sorted PV pairs (*x*-axis) for second GA search–outlier PV pairs labeled.

The largest positive difference (0.48) occurs in Figure 7 when the average user request timeout is 30 s. This is the same result found previously. This implies that if user timeouts are too short, then even with orphan-control procedures active, the cloud will fail to serve enough users, as the maximum anti-fitness was still 0.82, though the average decreased to about

 ISBN: 978-1-61208-308-7

0.55. Since orphan-control procedures operate over periods numbered in hours, virtual message losses caused by short user timeouts can still overtax the procedures. This finding was unknown previously. Another set of related parameters also exhibited large positive differences. For example, small standard deviations in user request timeouts tended to keep short user timeouts as short as possible. Short timeouts were exacerbated by increases in inter-site distances, especially when combined with short inter-router distances (i.e., more network hops between sites) and with higher simulated per-hop queuing delays. This implies that cloud designers must take wide latitude in considering many factors beyond their control that could determine the best user timeouts to encourage. On that issue, the PV with the largest negative difference (-0.24) was the user request timeout set to 120 s. This result implies that GA search can recommend optimal settings while simultaneously searching for failure scenarios. Finally, the iterated GA search also reestablished that small clusters would lead to overload and that impatient users could be a problem.

We conducted a third GA search over 209 generations. In that search, we changed the ranges of some parameter values to seek new failure scenarios and additional insights into system behavior. As expected, since we were searching for failure scenarios, the GA search found only slightly improved outcomes, yielding a maximum anti-fitness of 0.77 and an average of about 0.6. On the other hand, new insights were revealed. Figure 8 plots estimated probability differences for the 680 PV pairs found by the GA.

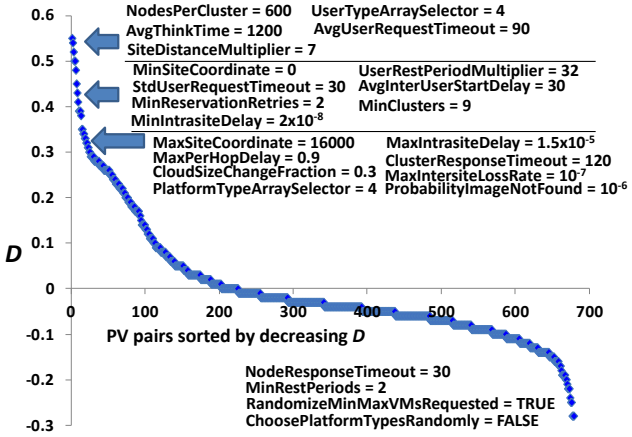

Figure 8. *D* (*y*-axis) for 680 sorted PV pairs (*x*-axis) for third GA search–outlier PV pairs labeled.

Though we increased (by 60 s) the range of user request timeouts, the new minimum timeout of 90 s proved to be too short, especially when coupled with specific network factors (such as long distances and large per-hop delays), along with 120 s delays by the cloud controller, when awaiting responses from clusters. Regarding response delays within the three-tiered cloud system, when the cloud waited only 60 s for cluster responses and clusters waited only 30 s for node responses, the system exhibited better outcomes. The user request timeout must accommodate delays due to network factors and timeouts within the cloud itself. The GA search found that an average user request timeout of at least 120 s (borderline outlier) was required to lower anti-fitness, and that 180 s (borderline outlier) provided the lowest anti-fitness.

Though we increased (by 400 nodes) the range of cluster sizes, a 600-node minimum size still proved too small. The GA found that at least 800 nodes per cluster were needed to avoid cloud overload for the parameters within the search space. Further, the GA discovered that a 30 s average inter-user startup delay, a parameter intended to gradually introduce load into the cloud, was too short, leading to cloud overload.

The GA found that homogeneous cluster sizes lowered anti-fitness, when compared with cases where 20% of clusters were large and 80% small. The GA also found that increasing and decreasing cloud size by 30% yielded higher anti-fitness than smaller size changes of 10% and 20%. Further, the GA found that cloud administrators needed to complete individual operations in a mean of 300 s (borderline outlier); 900 s (borderline outlier) was too long.

The GA also found insights related to platform types. First, assigning platform types randomly from a specified set (simulating a cloud constructed by adding any available nodes) increased anti-fitness. Second, one specific arrangement of platform types, where 28% of nodes had 32-bit architectures, increased anti-fitness when combined with simulated user types (60% web-service and 40% distributed-search applications) that required 64-bit architectures for all VMs.

All searches described above had the property that *H* subsets contained over 10^4 tuples, while comparable *L* subsets contained fewer than 10^3 tuples. This discrepancy in samples occurred naturally because the GA was searching for scenarios more likely to fall into *H* subsets. *L* subsets had as many as hundreds of tuples only because, as discussed previously, the low anti-fitness landscape of the Koala simulator was much larger than the high anti-fitness landscape. One could increase samples in *L* subsets by inverting the GA search to look for scenarios with low anti-fitness.

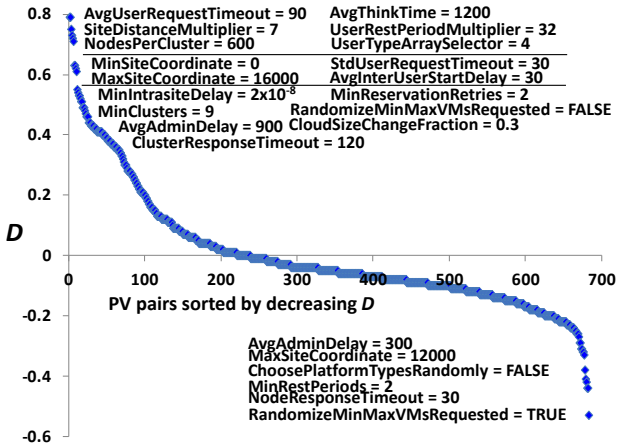

Figure 9. *D* (*y*-axis) for 683 sorted PV pairs (*x*-axis) for fourth GA search–outlier PV pairs labeled.

We inverted a fourth GA search. We used the same parameter space as in the third search, but instructed the GA to seek high fitness (i.e., low anti-fitness) scenarios. We ran the inverted search for 205 generations, and then combined the collected tuples with the tuples collected during the third search. After filtering, the resulting *H* subset contained 14601 tuples and the *L* subset contained 42253 tuples. Analysis of the

 ISBN: 978-1-61208-308-7

probability differences, shown in Figure 9, confirmed findings obtained from the third GA search.

VI. CONCLUSIONS AND FUTURE WORK

We defined a design-time method, combing GA search with simulation, to seek failure scenarios in system models. We applied the method in a case study, seeking (and finding) a known failure scenario in an existing IaaS cloud simulator. We iterated the GA search to reveal previously unknown failure scenarios. We used the case study to evaluate the dynamics, quality, effectiveness, and cost of GA search. Our GA searches explored predominantly non-duplicative scenarios with high anti-fitness. We uncovered evidence that GA search could reveal insights about optimal parameter settings, while simultaneously searching for failure scenarios. We also found that, due to high latency, GA search should be pursued only for systems with sufficient schedule time, and where failure scenarios have high cost.

We can extend our work in five directions. First, additional analysis methods need to be explored, to further mine the data collected by our GA searches. We can envision using statistical and information-theoretic techniques to extract features from the collected tuples, and then applying clustering algorithms to suggest specific classes of failure scenarios. Second, we should continue to explore our case study, attempting to uncover parameter subspaces where no failure scenarios can be found, and also using GA search under alternate definitions of anti-fitness to discover other kinds of system failure scenarios that might exist. Third, we should apply our method to models of other complex information systems, such as communication networks and other forms of computational clouds. This would allow us to confirm the generality of our approach. Fourth, we should seek partners, operating cloud computing systems or test beds, against which we can validate our method. Finally, we should investigate run-time methods to provide early signals of incipient failures. Such run-time methods are necessary because design-time methods are unlikely to discover all possible failure scenarios that could arise in a deployed system.

REFERENCES

[1] D. Takahashi, “Amazon’s outage in third day: debate over cloud computing’s future begins”, VB/News, April 23, 2011.

[2] Z. Michalewicz and D. Fogel, How to Solve It: Modern Heuristics, Springer, 2nd ed., 2004.

[3] D. Fogel (ed), Evolutionary computation: the fossil record, IEEE Press, 1998.

[4] M. Mitchell, An introduction to genetic algorithms, MIT Press, 1998.

[5] M. Fischer and J. Shortle, “Rare event simulation: enhancing efficiency, SigmaΣ noblis, 10:1, Sept. 2011, p. 52.

[6] C. Kelling and G. Hommel, “Rare event simulation with an adaptive “RESTART” method in a Petri net modeling environment”, Proceedings of the 4th WPDRTS, IEEE, Jan. 1996, pp. 229-235

[7] P. Ecuyer and B. Tuffin, “Splitting for rare-event simulation”, Proceedings of the Winter Simulation Conference, IEEE, Dec. 2006, pp. 137-148.

[8] D. Reijsbergen, P-T de Boer, W. Scheinhardt, and B. Haverkort, “Rare event simulation for highly dependable systems with fast repairs”, Performance Evaluation, 69:7-8, 2010, pp. 336-355.

[9] G. Galati, M. Naldi, and G. Pavan, “Stochastic simulation techniques as related to innovation in communications-navigation-surveillance and air traffic management”, Simulation Modeling Practice and Theory, 11, 2003, pp. 197-209.

[10] A. Shultz, J. Grefenstette, and K. DeJong, “Learning to break things: adaptive testing of intelligent controllers”, Handbook of Evolutionary Computation, Chapter G3.5, IOP Publishing Ltd and Oxford University Press, 1995, pp. 1-11.

[11] E. Yucesan and S. Jacobson, “Computational issues for accessibility in discrete event simulation”, ACM Transactions on Modeling and Computing Simulation, 6:1, 1996, pp. 53-75.

[12] A. Haines, K. Mills, and J Filliben, “Determining relative importance and best settings for genetic algorithm control parameters”, NIST Pub. #912472, Nov. 2012, pp 1-22.

[13] C. Dabrowski and F. Hunt, "Using Markov chains and graph theory concepts to analyze behavior in complex distributed systems", Proceedings of the 23rd European Modeling and Simulation Symposium, Sept. 2011, pp. 1-10.

[14] G. Fainekos, S. Sankaranarayanan, K Ueda, and H. Yazarel, "Verification of automotive control applications using S-TaLiRo," American Control Conference (ACC), Jun. 2012, pp. 3567-3572.

[15] Y. Matsuo, “Prediction, forecasting, and chance discovery”, Chapter 3 in Chance Discovery, Springer, 2003, pp. 30-42.

[16] F. Salfner, M. Lenk, and M. Malek, “A survey of online failure prediction methods”, ACM Computing Surveys, 42:3, 2010, Article 10, pp. 1-42.

[17] P. Gross et al., “Predicting electricity distribution feeder failures using machine learning susceptibility analysis”, Proceedings of the National Conference on Artificial Intelligence, 21:2, MIT Press, Jul. 2006, pp. 1705-1711.

[18] Q. Guan, Z. Zhang, and S. Fu, “Proactive failure management by integrated unsupervised and semi-supervised learning for dependable cloud systems”, 6th International Conference on Availability, Reliability and Security, Aug. 2011, pp. 83-90.

[19] F. Salfner, “Predicting failures with hidden Markov models”, Proceedings of 5th European Dependable Computing Conference, Apr. 2005, pp. 41-46.

[20] G. Weiss and H. Hirsh, “Learning to predict extremely rare events”, Proceedings of the AAAI Workshop on Learning from Imbalanced Data Sets, Jul. 2000, pp. 64-68.

[21] Q. Guan, Z. Zhang, and S. Fu. “Ensemble of bayesian predictors for autonomic failure management in cloud computing”, Proceedings of IEEE International Conference on Computer Communications and Networks, Jul. 2011, pp. 1-6.

[22] Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto, “Online failure prediction in cloud datacenters by real-time message pattern learning”, IEEE 4th International Conference on Cloud Computing Technology and Science, Dec. 2012, pp. 504-511.

[23] T. Chalermarrewong, T. Achalakul, and S. Wee See, "Failure prediction of data centers using time series and fault tree analysis”, Proceedings of the IEEE 18th International Conference on Parallel and Distributed Systems, Dec. 2012, pp.794-799.

[24] F. Salfner and P. Tröger, “Predicting cloud failures based on anomaly signal spreading”, 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Jun. 2012, pp. 1-2.

[25] C. Dabrowski and K. Mills, "VM leakage and orphan control in open-source clouds", Proceedings of IEEE CloudCom, Dec. 2011, pp. 554-559.

[26] K. Mills, J. Filliben, and C. Dabrowski, "An efficient sensitivity analysis method for large cloud simulations", Proceedings of the 4th International Cloud Computing Conference, IEEE, Jul. 2011, pp. 1-8.

[27] K. Mills, J. Filliben, and C. Dabrowski, "Comparing VM-placement algorithms for on-demand clouds", Proceedings of IEEE CloudCom, Dec. 2011, pp. 91-98.

 ISBN: 978-1-61208-308-7