A Reinforcement Learning Approach to Automatic Error Recovery

Qijun Zhu
Artificial Intelligence Lab
Tianjin University
Tianjin, China
zhuqijun@tju.edu.cn

Chun Yuan
Microsoft Research Asia
No.49, Zhichun Road
Beijing, China
cyuan@microsoft.com

Abstract

The increasing complexity of modern computer systems makes fault detection and localization prohibitively expensive, and therefore fast recovery from failures is becoming more and more important. A significant fraction of failures can be cured by executing specific repair actions, e.g. rebooting, even when the exact root causes are unknown. However, designing reasonable recovery policies to effectively schedule potential repair actions could be difficult and error prone. In this paper, we present a novel approach to automate recovery policy generation with Reinforcement Learning techniques. Based on the recovery history of the original user-defined policy, our method can learn a new, locally optimal policy that outperforms the original one. In our experimental work on data from a real cluster environment, we found that the automatically generated policy can save 10% of machine downtime.

1. Introduction

Maintaining high dependability has always been a critical topic for computer systems. Doing so usually is implemented in two ways: increasing reliability or availability. Reliability characterizes the ability of a system to perform services correctly, which can be measured by the meantime between failures (MTBF). Availability means that the system is available to perform services, which can be characterized by the meantime to repair (MTTR). Despite great improvements in research and practice in software engineering, latent bugs in complex software systems persist, and often it is just too difficult to improve system reliability by recognizing faults or fixing bugs. Actually, as the complexity of the software systems increases dramatically, analyzing system problems and finding root causes has become costly and time-consuming work even for skilled operators and diagnosticians [13][18]. Making computer systems more consistently available is indeed practical and can increase effectiveness and productivity.

Traditional fault tolerant techniques rely on some form of redundancy to achieve high availability, which can come in the form of function or data redundancy. However, such methods usually sacrifice system performance and can cause high hardware costs and increase complexity. For example, process pairs [3] utilize good processors taking over the functionality of failed processors in which non-stop processing is at the cost of hardware redundancy and performance. Auragen [4] also applies a similar scheme to the UNIX environment.

Another important way to achieve high availability is through recovery schemes that restore systems to a valid state after a failure. One of these recovery schemes is based on check-pointing, which periodically creates a valid snapshot of a system's state and, in the case of a failure, returns the system to a valid state. Often this method is system-specific and may create great burdens on system designers and operators. Baker *et al.* [1] utilized Recovery Box to realize quick recovery in which operating systems and application programs need to use the interface provided by Recovery Box to implement data insertion and retrieval. Moreover, it is difficult to determine the right time to create a checkpoint and ensure its validity.

A more popular recovery scheme is simple rebooting technique, which can be applied at various levels and is employed by many nontrivial systems today. Actually, a significant fraction [5][10][14][22] of failures are cured by simple recovery mechanisms such as rebooting even when exact causes are unknown. Candea *et al.* [6] built crash-only programs to crash safely and recover quickly, and then improved this approach by introducing a fine-grained mechanism called microreboot [7] which can provide better recovery performance and cause less disruption or downtime.

However, to achieve efficient error recovery, potential repair actions need to be scheduled reasonably based on policies like state-action rules. An example of such policies includes recursively attempting the remaining cheapest action [7]. This simple policy may not be sufficient in real environments because of imprecise fault localization, recurring failures, or failed repair actions [8]. The overall cost of cheap actions, including the time for observing recovery effects, is actually not that negligible either. Due to similar difficulties in root cause analysis, as mentioned above, generating recovery policies automatically could be important in effective error recovery. Joshi *et al.* [16][17] attempt to tackle the problem with a model-based approach that enables automatic recovery in distributed systems. Though their method works well in simulated experiments, there are still problems. First, the method needs detailed information on the system model, which is often too complex to obtain for large-scale systems. Second, it can locate faults well along the recovery process, but may have difficulties in determining how to deal with the faults since some faults may need the combination of several actions to complete a recovery in real systems.

In this paper, we also utilize application-independent techniques to achieve automatic recovery. However, we are focusing on recovery policy generation when system models are not available. To the best of our knowledge, this has not been fully studied before. We have investigated how to make proper decisions on which repair action to choose when the actual root cause is only localized at a coarse level. Particularly, we propose a novel approach based on reinforcement learning (RL) to automatically find the locally optimal policy, and show that it can achieve better recovery performance. Another benefit of our learning-based approach is that it can adapt to the change of the environment without human involvement.

Our contributions are as follows:

1. An offline reinforcement learning method to automatically generate optimal recovery rules. We should point out that the generated rules are locally optimal since the learning is restricted by the original, user-defined rules to be optimized.
2. A hybrid approach to handle noisy states that cannot be cured by generated rules. The results show that our approach cannot only maintain nearly the same performance as using the generated rules in isolation, but also can cover all possible states.
3. A new type-oriented model of automatic error recovery. Each rule corresponds to a potential error type induced from the recovery log.
4. Some experience in reducing rule-training time. By using a selection tree, we can guarantee discovery of optimal rules within much less time than the standard RL process.

The rest of the paper is organized as follows. Section 2 defines the automatic recovery problem and provides an overview of our approach. Section 3 gives additional details on the training method, and presents some assumptions based on how a reasonable evaluation cab be conducted. Section 4 describes our experimental data and evaluation framework. Section 5 presents experimental results. Section 6 discusses related work and Section 7 serves as our conclusion.

2. Overview

An automatic recovery framework typically consists of three functions: event monitoring, fault detection, and error recovery, as shown in the upper part of Figure 1. A recovery process may run like the following: Event monitoring collects various information and events for further analysis, such as symptoms of error states corresponding to different faults that occur in the target system. Then, fault detection recognizes failures and informs error recovery so that it can decide which repair action should be used based on the given recovery policy and the failure information. The chosen action is applied to the corresponding component and the result of the recovery will be monitored, which may lead to another round of recovery.

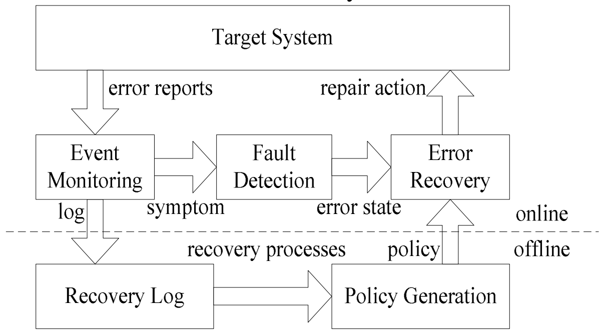


Figure 1: Automatic recovery framework.

Usually, recovery policies are user-defined by system developers or operators. The issues with this approach are manifold. First, policies are often difficult to build and evaluate for large-scale, complex systems in which detailed system models may not always be available or up to date. Second, an ideal policy should be able to target each fault. However, due to the limitation of fault localization, people often have to build coarse-grained policies to cover all possible error states. This sometimes may be too inaccurate to guarantee the desired result. Third, unanticipated errors and varying symptoms may appear throughout the running of systems, which requires policies evolve over time.

In our recovery framework we have two additional offline components for automatically generating recovery policies, as shown in the lower part of Figure 1. Recovery log keep a history of error recovery via the event monitoring component. Policy generation components learn recovery policies from the recovery history with statistical induction techniques to instruct error recovery. Specifically, we use reinforcement learning to generate error type-oriented policies. Our simulated experiments show that the policies learned by our method outperform manual ones.

2.1. Problem formalization

If we consider the recovery process as selecting a repair action according to current state and then getting a reward (e.g. recovery time) after taking the action, we can naturally formalize it as a sequential decision-making process, or particularly a Markov decision process (MDP) [15].

A Markov decision process can be represented as a tuple (S, A, δ, c, S_0), where S is the set of possible error states, each of which consists of some related features; In particular, S_0 is all possible starting states; A is the available repair actions; δ is the state transition function, which decides the next state s_{t+1} based on the current state s_t and the selected action a_t; c is the cost function, which determines the cost for executing an action under an error state. In our experiments, we use Meantime to Repair (MTTR) as the metric for evaluation, so c is based on recovery time (downtime). Therefore our goal is to minimize the expected cost V,

$$V = E[\sum_{t=0}^{\infty} c(s_t, a_t)] \tag{1}$$

that is, to achieve the shortest recovery time. We will give more detailed explanation in Section 3.2.

2.2. Reinforcement learning and Q-learning

As further background of our method, we give a brief introduction to reinforcement learning, an unsupervised learning method for sequential decision making. In this learning paradigm, the learning agent receives reinforcement (reward) after each action execution. The objective of learning is to construct a control policy so as to minimize the discounted cumulative reinforcement in the future or, for short, utility:

$$V_t = \sum_{k=0}^{\infty} \gamma^k c_{t+k} \tag{2}$$

which is a generalized form of equation (1). γ is the discount factor. In this paper, we simply set it to 1.0 to make sure the expected cost is equal to MTTR.

Q-learning is a widely used reinforcement learning algorithm. The idea of Q-learning is to construct an evaluation function called Q-function,

$$Q(state, action) \rightarrow utility$$

to predict the utility when the agent is executing some action in certain state. Given an optimal Q-function and a state s, the optimal control policy is simply to choose the action a such that $Q(s, a)$ is minimal over all actions. Often the Q-function can be represented in a generalized way like multi-layer neural networks and incrementally learned through temporal difference (TD) methods [23]. Given a sequence of state transitions, the Q-function can be computed by iteratively applying the learning procedure to each two successive states along the sequence. Note that this procedure is actually the simplest form of TD methods, $TD(0)$. More details and discussions on Q-learning algorithm can be found in standard machine learning textbooks or related papers [20][21].

2.3. Automated policy generation

In this section, we will present the motivation for offline training and a brief description of the policy generation process.

2.3.1. Offline training. There are a few issues in applying reinforcement learning to learn recovery policies online.

1. Before finding out the optimal policy, the RL training process may explore many bad policies, which, once applied, might seriously degrade normal system performance.
2. The training process may start with an arbitrarily bad policy.
3. The training process requires tens of thousands of observations. For error recovery, several years may be required to converge for infrequent errors.

To address these limitations, we devised an offline training method that enables RL to take advantage of user-defined policies. Although it is at the cost of missing the globally optimal policy and only producing the locally optimal one, the obvious improvement it can bring to original policies and the avoidance of online training overhead still makes it a reasonable choice.

2.3.2. RL approach. We use the error types induced from failure symptoms to approximate the real faults. An induced error type represents the errors that share the same symptoms, which ideally corresponds to a unique fault, though different faults may be inferred as the same error type. Specifically, we simply use the error types and the previously tried actions to form the states. The learning algorithm analyzes a real-world recovery log generated by a user-defined policy and computes the value of the Q-function $Q(s, a)$, which satisfies the following equation

$$Q(s, a) = E[c(s, a)] + \sum_{s'} P(s'|s, a) min_{a'} Q(s', a')$$

$$\text{, where } s' = \delta(s, a) \tag{3}$$

Here, the Q-function $Q(s,a)$ stands for the minimal time cost for state s beginning with action a. The generated recovery policy may be restricted by two factors, the error types and the original recovery policy, so it can only achieve local optimum.

3. Approach

This section gives details on the RL approach to automatic recovery policy generation, and discusses some difficulties and our solutions.

```
function Q-learning
input
        Pr   recovery processes (in the recovery log)
        Q    initial Q-function values
return
        updated Q-function values
begin
    // select one recovery process from Pr
    p = SelectProcess(Pr)
    // induce error type based on recovery process
    t = InduceErrorType(p)
    // build initial state
    s = InitialState(t)
    // explore different recovery actions
    while(!Healthy(s)){
        a = SelectRecoveryAction(Q, s)
        c = UpdateState(Pr, s, a, s')
        Record(s, a, c, s')
        s = s'
    }
    // update Q-function values
    for every two successive states s, s' in record
        UpdateQfunction(Q, s, s')
    return Q
end
```

Figure 2: Q-learning algorithm for optimal policy generation

As stated in the previous section, we use Q-learning algorithm to obtain repair policy. The training process is implemented by applying Q-learning algorithm to each error type, which can be inferred from error symptoms in the recovery log. The procedure described in Figure 2 is iteratively used on the recovery log to get an optimal Q-function. In the following sections, we provide a closer look at each key step.

3.1. Error type inference and noise filtering

In this paper, we attempt to extract potential faults based on the error symptoms in the recovery log.

To get a rough idea of how symptoms are distributed, we generate a number of symptom sets from a real-world recovery log (to be introduced in section 4.1). In each set, the symptoms are highly related based on the ratio of the number of recovery processes in which they appear together out of all the recovery processes in which one symptom appears. Due to the fact that some symptoms may occur quite infrequently, we use m-pattern algorithm [19], which is capable of finding infrequent but highly correlated items, to mine mutually dependent symptoms in the log. The strength of mutual dependence is measured by parameter *minp*.

We summarize the percentage of the recovery processes with only highly dependent symptoms for various dependence strength in Figure 3. We can observe that the whole log is mainly made up of a number of highly cohesive symptom sets. Additionally, we find different sets share few intersections. This motivates us to generate policy at symptom level since we do not have any knowledge about real faults. Actually, we think the symptom sets may have strong correlation with the faults in the system. Based on these observations, we define *error type* as the initial symptom of a recovery process to approximate the real fault. For example, if the sequence of symptoms occurring during a recovery process is “A; B; C”, then we use symptom “A” to represent its error type. We choose the initial symptom since it is usually representative enough of the symptom set to which it belongs and the other symptoms in the recovery process often co-occur with it. Based on this definition, we employ the error type as the unit in building recovery policies.

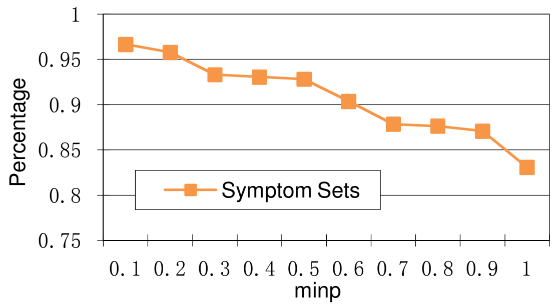


Figure 3: Symptom sets extracted from recovery log

Moreover, we still need do some noise filtering based on the above results because the evaluation is based on a simulation platform in our experiments and those noise data are often too difficult to simulate and may impact the precision of the evaluation. Actually we choose *minp* = 0.1 in m-pattern algorithm, and ultimately get 119 symptom clusters covering 96.67 % of the total logs. The left 3.33% are regarded as noisy cases that may contain more than one error. The noise data only take up a trivial part within the logs, so the filtering process does not influence the conclusions much. Although our RL approach can also be applied to these noisy cases, we still ignore them to get a precise evaluation.

3.2. State transition

We use error types (beginning symptoms) and previously tried repair actions to define states. A state s_t is represented by a tuple $(e, r, (a_o,\ a_1, \dots, a_{t-1}))$, where e is error type, r is the recovery result (failure or health) before time t, and a_i, $i = 0, 1, \dots, t-1$, are all repair actions executed before. From this definition, it is obvious that before the last repair action the recovery result r of any state will be f (failure) and after that it will become h (health). This definition also makes automatic error recovery a Markov decision process.

Transition function, δ, here is partially known, since the state s_{t+1} produced by the acts on $s_t = (e, f, (a_0, a_1, \dots, a_{t-1}))$ and a_t could only be two types, $s_{t+1}^f = (e, f, (a_0, a_1, \dots, a_{t-1}, a_t))$ or $s_{t+1}^h = (e, h, (a_0, a_1, \dots, a_{t-1}, a_t))$, the probabilities of which depend on the environment and properties of the errors. So the equation (3) could be rewritten as

$$Q(s_t, a_t) = E[c(s_t, a_t)] + q(s_t, a_t) \min_{a_{t+1}} Q(s_{t+1}^f, a_{t+1})$$
$$\text{,where } s_{t+1}^f = \delta(s_t, a_t) \qquad (4)$$

Figure 4 illustrates the decisions and possible sequences in a recovery process after an error is detected.

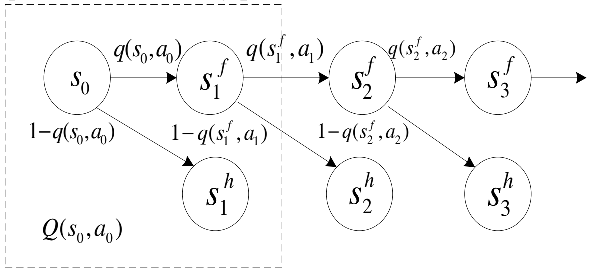


Figure 4: Error recovery process. The Q-function $Q(s, a)$ is the expected time cost for both two directions (failure or health).

We restrict the count of the repair actions to a finite number N for each recovery process (in the experiment we set N = 20). It means that if the preceding N-1 repair actions fail to cure the problem, we will end the process by requesting a manual repair (the recovery action which is to be conducted by human). Since all policies produced with this limitation are proper, according to the theorem of value contraction in [14], our RL method will converge with probability 1.

3.3. Explore strategy and table update

To explore different repair actions, we first need to infer the state transitions based on the existing recovery processes. This amounts to finding the correct repair actions for each recovery process. The easiest method is to choose the last repair action as the correct one. However, it may not always be safe to make such assumptions and sometimes some stronger repair actions also play an important role in the recovery process. A more realistic assumption is to regard both the last action and other stronger actions as the correct repair actions. Besides, since a stronger action includes the processes of the weaker ones, it can at least cause the same effect as the weaker ones. Based on this analysis, our hypotheses about the recovery process is as follows:

1. For any successful recovery process, we need at least the same correct repair actions (including the last action and the stronger ones in the process) to achieve the same recovery result.
2. Stronger actions can replace weaker ones in a successful recovery process.
3. Recovery processes for different errors are independent of each other.

With these hypotheses, we can carry out the explore strategy and estimate the time cost for each possible policy.

Starting from some initial states of recovery processes, we have to explore a large enough state space first and then find the optimal policy. We can roughly divide the learning course into two phases, one for exploration and the other for search. Like the simulated annealing algorithm, we use a temperature T to control the learning course from exploration to search.

Actually, at time t for certain error e, we will utilize the following probability distribution (Boltzmann distribution) to select a repair action stochastically,

$$P(a_i|s_t) = \frac{e^{-\frac{Q(s_t,a_i)}{T}}}{\sum_j e^{-\frac{Q(s_t,a_j)}{T}}},\quad a_i \in A \qquad (5)$$

Here, the temperature T will decrease with more and more recovery processes analyzed, so the repair action will eventually be selected completely based on Q values, thus generating the policy.

When a repair action is selected, its time cost will be estimated based on the recovery log. Specifically, one of the following values will be chosen: actual time cost in the recovery process average success time cost, or average failing time cost. Based on these values, we can further update the Q-function and reasonably evaluate the policy. As we show in Section 4.2, this approach works well in our experiments.

Another important step in the whole training course is how to update the Q values. In our method, we chose to use a table look-up representation of the Q-function and update the Q values based on the following equation,

$$Q_n(s,a) \leftarrow (1 - \partial_n) Q_{n-1}(s,a)$$
$$+ \partial_n \left[c(s,a) + \min_{a'} Q_{n-1}(s', a')\right]$$
$$\text{and, } \partial_n = \frac{1}{Visits_n(s,a)} \qquad (6)$$

where $Q(s,a)$ records the expected value of the Q-function, and $Visits(s,a)$ represents how many times (s,a) pair is explored. It is easy to prove that this updating method is contracted and Q values will eventually converge to the optimal ones [20].

3.4. Hybrid approach

Occasionally, the RL-trained policy might fail to repair some exceptional error cases. To get beyond this issue, we provide a hybrid approach that combines the trained policy with the user-defined one. In particular, if an error still exists after the last action selected according to the trained policy, we will automatically revert to the user-defined policy. Since these noisy cases do not happen frequently, the hybrid policy cannot only guarantee to repair all errors as well as the user-defined policy does, but also can maintain the advantage of automatic policy generation with RL, as we show in Section 5.2.

4. Experimental setup

This section introduces the data used in our experiments and the simulation platform that outputs feedback of a repair action on a state -based on the hypotheses.

4.1. Experimental data

Our experimental data are based on the recovery log collected from a large-scale cluster system with thousands of servers that contained more than 2 million entries of error symptoms and repair actions over nearly half a year of operations. The recovery policy used in the real system is user-defined, which mainly tries the cheapest action enabled by the state. There are four actions for repairing a machine: TRYNOP (simply watch and do not try any operation), REBOOT, REIMAGE (rebuild the operating system), and RMA (let human repair).

Table 1: Example of recovery process (machine name is omitted).

Time	Description (details omitted)
3:07:12 am	error:IFM-ISNWatchdog: …
3:10:58 am	errorHardware:EventLog: ...
3:23:26 am	TRYNOP
3:25:37 am	errorHardware:EventLog: ...
3:27:34 am	errorHardware:EventLog: ...
3:42:10 am	REBOOT
4:13:07 am	Success

The log entries can be represented in the format of <time, machine name, description>, in which the description can be the repair action, symptom of an error, or report of a successful recovery that occurs at the recorded time on the monitored machine. Therefore, the logs can be divided into an ensemble of recovery processes. The processes start with the advent of a new error, experience a series of repair actions, and end with successful recovery. Table 1 gives an example of recovery process.

After noise filtering, we get 97 error types from the recovery log with the error type inference method mentioned in Section 3.1. To guarantee enough training data, we choose the 40 most frequent error types, which constitute 98.68% of the total recovery processes.

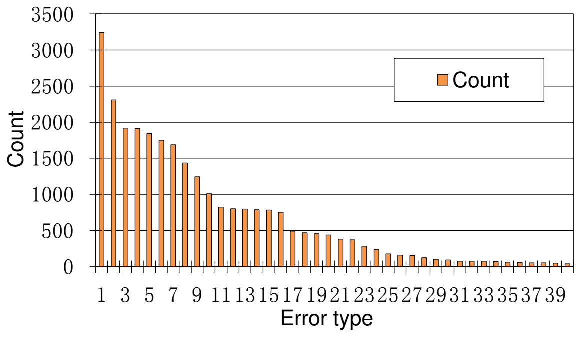


Figure 5: Count of 40 most frequent error types

Figure 5 shows the count of the selected error types. The remaining error types, since they are much less frequent, still need more time to accumulate enough training samples by monitoring the real system. The total downtime of each error type in the recovery processes controlled by the user-defined policy is given in Figure 6.

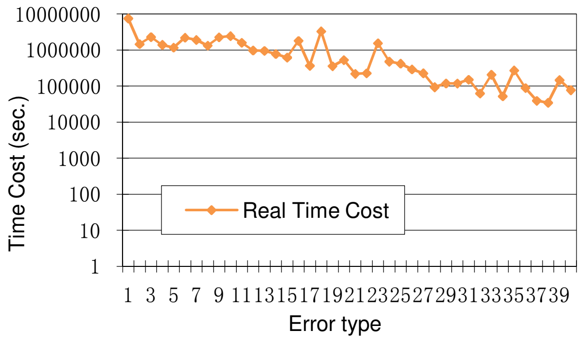


Figure 6: Total downtime of 40 most frequent error types under user-defined policy

4.2. Simulation platform

Our simulation platform is built to compute time cost for a repair action on a state based on the assumptions mentioned in Section 3.3 and the recovery log.

To verify our assumptions and the settings of the simulation platform, we run the platform under the user-defined recovery policy of the real system. Because we could not refer to all the information considered by the user-defined policy from the log, we could only expect an approximate result. Figure 7 shows the results for the 40 most frequent error types. The relative time cost here is the ratio of the estimated time cost compared to the real one for each error type, which is also used as the evaluation measure in the following experiments.

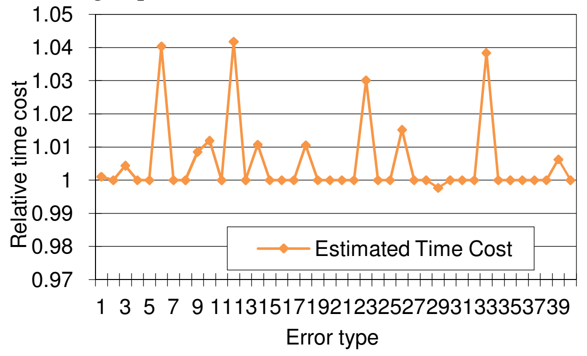


Figure 7: Relative cost for 40 most frequent errors compared to real ones. Biggest deviation is less than 5%.

We can see that the time costs computed by the simulation platform are close to the real ones and only one computed cost (error type 29) is slightly less than the real one. Therefore, by using this platform we can expect a conservative evaluation for most cases and thus make a fair comparison between the trained policy and the original policy.

5. Experimental results

In this section, we first evaluate the policy originally trained by RL, then the performance of the hybrid approach. In each experiment we will apply the learning algorithm to a portion of the log to train a policy, and then test the performance of the policy on the remaining log. The training set and the test set are divided according to time order. We choose 20%, 40%, 60% and 80% of the recovery log for used in the training, thus forming four tests (test 1, test 2, test 3, and test 4).

5.1. Results of RL-trained policy

Figure 8 shows the fractions of the estimated time cost of the trained policy with respect to the actual time cost for each error type. The time cost of the unhandled cases is not counted in the total cost.

In Figure 8, the four plots show the results of the four policies trained with 20, 40, 60 and 80 percent of the whole log. For most error types, the trained policy performs almost the same as the original policy. Through our observation of the corresponding recovery log, we find that the original policy has already achieved good enough recovery steps. This is hard to optimize any more based only on the existing log. On the other hand, we find that for some error types such as 1, 35, and 39, the trained policy gains a significant improvement over the original policy, reducing the cost to nearly half. When looking at the policy more closely, we find that the trained policy for most error types is nearly the same as the original one. The deviation of the time cost for some error types (e.g. 6, 10, and 23) comes from simulation error (see Section 3.1). For error type 1, 35, and 39, the trained policy will try a stronger repair action at the beginning instead of the weakest one as done by the original policy. Since the stronger action is more effective in recovering the system from the error, it gains a big savings in recovery time without trying the weaker actions first and waiting to find out that they do not work.

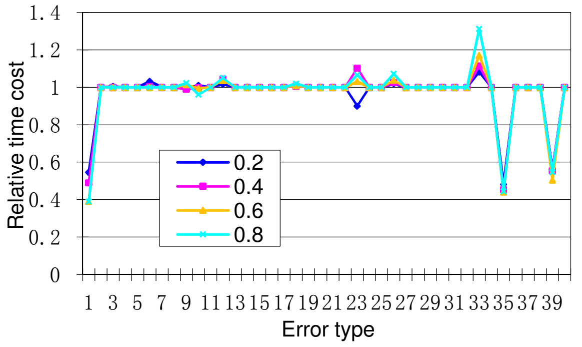


Figure 8: Relative time cost for trained policy compared to real one

The overall absolute time cost for the different test sets is shown in Figure 9. We can see that the trained policy can always gain over 10% time savings in the four tests. In particular, the policy trained from 40% of the log results in only 89.02% of the original downtime on the remaining log. Here, we only summarize the total time cost of the cases that could be handled by our trained policy. Since some unhandled cases exist that will be discussed in the next paragraph, the total time cost is a little less than the following experiment.

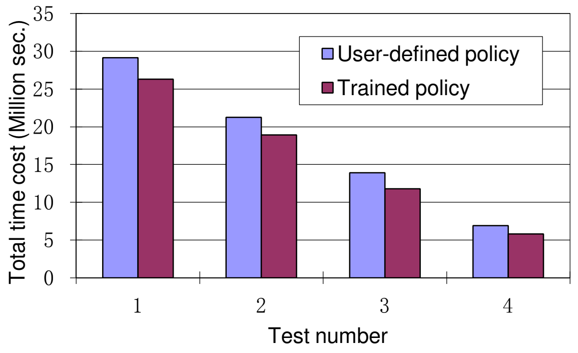


Figure 9: Total time cost of trained policy under different tests

Finally, we present the *coverage* of the trained policy in Figure 10, that is, the percentage of the errors it can handle. For each test, only a small number of error types cannot be handled and even in these cases the coverage is still more than 90%. Besides, the unhandled cases decrease dramatically with the more training data used.

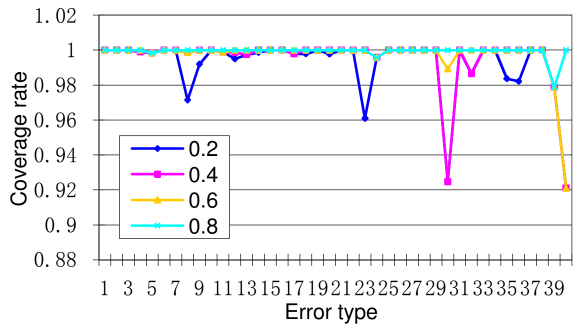
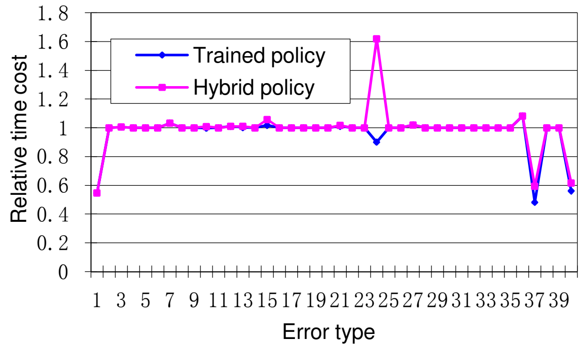


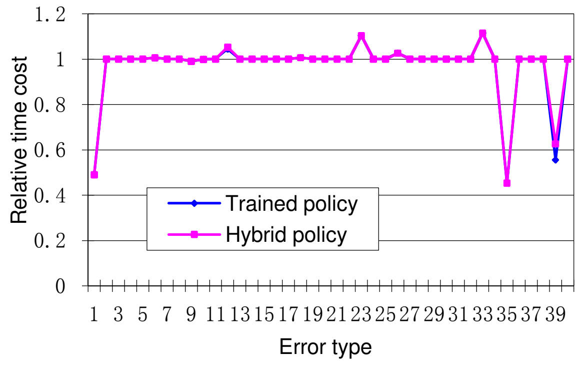
Figure 10: Coverage of the trained policy.

5.2. Results of hybrid approach

To solve the noisy cases not covered by the RL-trained policy, we combine it with the original user-defined defined policy. Figure 11 shows two results comparing the pure RL approach and the hybrid approach.



(a) Training set proportion = 0.2



(b) Training set proportion = 0.4

Figure 11: Performance comparison between trained policy and hybrid policy

For the policy trained with 20% of the log, the performance of the hybrid approach is almost the same as the RL-trained policy except for several exceptions, such as error type 23 (Figure 11(a)). However, when we take a closer look at the training data of error type 23, we find that some new patterns that appear in the test set are not covered by the training set, so the trained policy is suboptimal and may not perform stably. As the size of the training data increases, more precise policy is generated and the hybrid approach performs nearly the same as the trained policy, as presented in Figure 11(b).

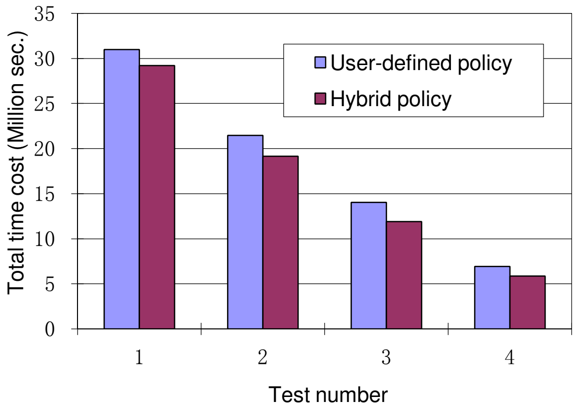


Figure 12: Total time cost of hybrid approach under different tests

Figure 12 summarizes the total cost for the original user-defined policy and the hybrid policy. Like the trained policy, the hybrid policy can also achieve more than 10% improvement over the original policy, on average. Corresponding to the policy trained with 40% of the log, the hybrid approach only costs 89.18% of the original downtime.

5.3. Learning rate experience

In this section, we introduce our effort in improving the learning process to shorten the training time. To this end, we use a technique called selection tree in the learning process. To build the selection tree, we consider the best two repair actions each time when generating the policy from the Q values. If the expected total cost of the second best action is close enough (based on a threshold) to that of the best one, we will choose both actions as candidates. Otherwise we will only choose the best one. Then, the selection tree can be built by iteratively putting these candidate actions as the children of the previous repair action, and the optimal policy can be generated by scanning the tree. Figure 13 shows the training time of this method (with selection tree) compared to the standard RL training course (without selection tree) with a maximum of 160 thousand sweeps (training set proportion = 0.4).

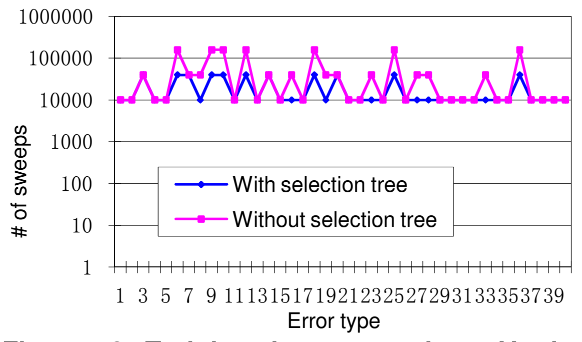


Figure 13: Training time comparison. Vertical axis stands for sweep number before convergence of training process for each error type.

Moreover, the performance of the policies trained by these two methods is shown in Figure 14. We can find that, using standard RL method, some training courses do not converge to the optimal policies even after 160 thousand sweeps. In contrast, with a selection tree, we can speed up the learning rate and successfully find the optimal policy within 40 thousand sweeps in our experiment.

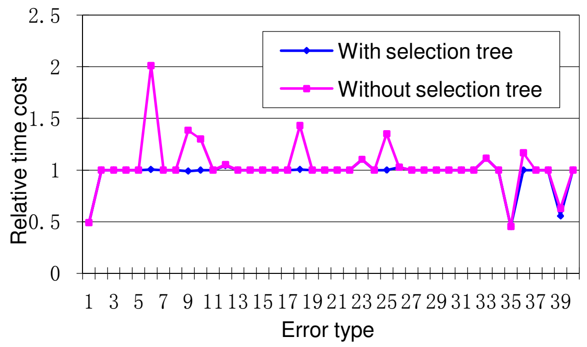


Figure 14: Performance comparison between optimized training method and standard method

6. Related work

Gray's classic text on failure analysis [14] surveys a range of failure statistics of a commercially available fault-tolerant system. He also discusses various approaches to software fault-tolerance, which mainly focus on how to prevent the occurrence of failures or reduce their frequency.

More recent work attempted to employ statistical learning techniques in automated fault diagnosis and performance management. An early work by Ma and Hellerstein [19] presented an efficient algorithm to discover all infrequent Mutually Dependent Patterns (*m-pattern*) for system management, for example, isolating problems in computer networks. Some statistical tools were developed for diagnosing configuration errors that cause a system to function incorrectly. Whitaker et al. presented the *Chronus* tool [26], which automates the task of searching for a failure-inducing state change. A similar work was completed by Wang et al. [25]. They presented the PeerPressure troubleshooting system that uses statistics from a set of sample machines to diagnose misconfiguration on a sick machine. There are still many projects focusing on performance analysis and debugging or bottleneck detection. Examples that include Magpie [2] and Pinpoint [9] make efforts to associate failures or performance problems with possible components via request traces. On failure diagnosis, Cohen *et al.* proposed to correlate the low-level system metrics with high-level performance states using Tree-Augmented Naïve Bayesian networks [11]. Based on this work, they construct *signatures* for clustering and retrieving, which could yield insights into the causes of observed performance effects and provide a way to leverage past diagnostic efforts [12][28]. Yuan et al. [27] proposed to correlate known faults to system behaviors with pattern classification techniques so as to recognize future occurrences of the faults automatically. Compared to these research efforts, our approach focuses on automated error recovery instead of performance diagnosis. Recently, Tesauro *et al.* [24] completed a similar work that employed a hybrid reinforcement learning approach to performance management.

In the area of error recovery, the common approaches rely on a priori knowledge from human experts to build policies for systems. Often, these repair actions can be expensive, causing nontrivial service disruption or downtime. Some recent research seeks to improve this method by introducing fine-grained recovery mechanisms. Microreboot [7], for example, provides a way for recovering faulty application components in an Internet auction system, without disturbing the rest of the application. We believe this work is complement to our work since we do not set any limitations on the set of repair actions. Additionally, with more potential repair actions, authoring or generating reasonable recovery policy will become evermore critical.

7. Conclusion

In this paper, we proposed a novel reinforcement learning approach to improve the framework of automatic error recovery. Specifically, we focus on recovery policy generation when a system model is not available, which we believe has not yet been fully studied. We have investigated how to make proper decisions on which repair actions to choose when the actual root cause is only localized at a coarse level. With our method, a locally optimal policy is guaranteed to be found, and it can adapt to changes in environment without human involvement. Finally, experimental results on data from a real cluster environment show that automatically generated policy achieved more than 10% savings in machine downtime on average. Several possible extensions of the approach include using generalization functions to approximate the Q-learning values, introducing more complicated relationships among actions, and designing initial policies that can be improved. In addition, we believe the approach provide greater benefits when we gain more information from event monitoring or fault detection.

8. Acknowledgments

We would like to thank Ken Cao, Peirong Liu and Yi Li for clarifying the details of the user-defined policy and the recovery log. We also thank the anonymous reviewers for their helpful comments and Dwight Daniels for proofreading the paper.

References

[1] M. Baker and M. Sullivan. The Recovery Box: Using fast recovery to provide high availability in the UNIX environment. In Proc. Summer USENIX Technical Conference, San Antonio, TX, 1992

[2] P. Barham, A. Donnelly, R. Isaacs and R. Mortier. Using Magpie for request extraction and workload modeling. In Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI), Dec. 2004.

[3] J.F. Bartlett. A NonStop kernel. In Proc. 8th ACM Symposium on Operating Systems Principles, Pacific Grove, CA, 1981.

[4] A. Borg, W. Blau, W. Graetsch, F. Herrman and W. Oberle. Fault Tolerance under UNIX. ACM Transactions on Computer Systems, 7(1): 1–24, Feb 1989.

[5] E. Brewer. Lessons from giant-scale services. IEEE Internet Computing, 5(4):46–55, July 2001.

[6] G. Candea and A. Fox. Crash-only software. In Proc. 9th Workshop on Hot Topics in Operating Systems, Lihue, Hawaii, 2003.

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman and A. Fox. Microreboot – A Technique for Cheap Recovery. In Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI), Dec 2004.

[8] G. Candea, E. Kiciman, S. Kawamoto and A. Fox, Autonomous Recovery in Componentized Internet Applications. Cluster Computing Journal, 9(1), Feb 2006

[9] M. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer. Pinpoint: Problem determination in large, dynamic systems. In Proc. 2002 Intl. Conf. on Dependable Systems and Networks, Washington, DC, June 2002.

[10] T.C. Chou. Beyond fault tolerance. IEEE Computer, 30(4):31–36, 1997.

[11] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons and J.S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In Proc. 6th Symposium on Operating Systems Design and Implementation, Dec. 2004.

[12] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), Oct. 2005.

[13] A. Fox and D. Patterson. Self-repairing computers. Scientific American, June 2003.

[14] J. Gray. Why Do Computers Stop and What Can Be Done About It? 6th International Conference on Reliability and Distributed Databases, June 1987.

[15] G.J. Gordon. Stable Function Approximation in Dynamic Programming, tech report CMU-CS-95-103, 1995.

[16] K.R. Joshi, W.H. Sanders, M.A. Hiltunen and R.D. Schlichting. Automatic Model-Driven Recovery in Distributed Systems. SRDS 2005: 25-38

[17] K.R. Joshi, W.H. Sanders, M.A. Hiltunen and R.D. Schlichting. Automatic Recovery Using Bounded Partially Observable Markov Decision Processes. In Proc. of the 2006 International Conference on Dependable Systems and Networks (DSN'06): 445-456

[18] J.O. Kephart and D.M. Chess. The vision of autonomic computing. Computer, 36(1):41–50, 2003.

[19] S. Ma and J.L. Hellerstein. Mining Mutually Dependent Patterns for System Management. IEEE Journal on Selected Areas in Communications, VOL. 20, NO. 4, May 2002.

[20] T.M. Mitchell. Machine Learning. McGraw-Hill, 1997.

[21] S.A. Murphy. A Generalization Error for Q-Learning. Journal of Machine Learning Research, 6 (2005) 1073–1097.

[22] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. Quality and Reliability Engineering Intl., 11:341–353, 1995.

[23] R.S. Sutton. Learning to Predict by the Methods of Temporal Differences. Machine Learning 3: 9-44, 1988.

[24] G. Tesauro, R. Das and N. Jong. Online Performance Management Using Hybrid Reinforcement Learning. First Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML'06), June 2006.

[25] H.J. Wang, J.C. Platt, Y. Chen, R. Zhang and Y.M. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In Proc. 6th Symposium on Operating Systems Design and Implementation, Dec. 2004.

[26] A. Whitaker, R.S. Cox and S.D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In Proc. 6th Symposium on Operating Systems Design and Implementation, Dec. 2004.

[27] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang and W.-Y. Ma. Automated Known Problem Diagnosis with Event Traces. 1st EuroSys Conference, April 2006

[28] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons and A. Fox. Ensembles of Models for Automated Diagnosis of System Performance Problems. In Proc. of the 2005 International Conference on Dependable Systems and Networks (DSN'05).