Scalable Rollback for Cloud Operations using AI Planning

Suhrid Satyal, Ingo Weber, Len Bass, Min Fu
NICTA, Sydney
{Suhrid.Satyal, Ingo.Weber, Len.Bass, Min.Fu}@nicta.com.au

Abstract—**Human-induced faults play a large role in systems reliability. In cloud platforms, system administrators may inadvertently make catastrophic mistakes, like deleting a virtual disk with important data. Providing rollback for cloud operations can reduce the severity and impact of such mistakes, by allowing to revert back to a known, good state.**

In this paper, we present a scalable approach to rollback operations that change state of a system on proprietary cloud platforms. In our previous work, we provided a system that augments cloud APIs and provides rollback operation using an AI planner [1]. However, the previous system eventually suffers from the exponential complexity inherent to AI planning tasks. In this paper, we divide and parallelize rollback plan generation, based on characteristics unique to the rollback scenario. Through experimental evaluation, we show that this approach scales better than the previous, naïve approach, and effectively avoids the exponential behavior.

Keywords*—*reliability, AI planning, cloud computing, web service

I. Introduction

For cloud applications, especially at large scale, one of the primary impediments to system dependability is human error. Human operator error is attributed with being the root cause of 20-50% of system outages [2], [3]. In many cases of outages caused by human operators, they may not be able to respond quickly enough to minimize the impact (e.g. losing traffic, violating SLAs, etc). Moreover, operators may perform an action that cannot be fixed entirely, like irreversibly deleting a data store. Such results can be caused with a single API call on cloud platforms like the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) platform [4].

In the context of API controlled cloud platforms, operators risk creating outages by executing undesired actions or actions whose effects are not fully known to them. Once an undesired state of the cloud resources has been reached, it is not necessarily clear to the operator how to revert back to an earlier, good state, for the following reasons. APIs from public cloud providers are provided *as-is*, and correcting mistakes is only possible by calling the right API operations in the right sequence. In some cases, API operations are irreversible, and no combination of actions can completely recover from the mistake (e.g. deleting a virtual disk).

One way of mitigating human errors is by designing platform that allows administrators to rollback their changes. However, in an API-controlled proprietary system this is not feasible: the operator does not have a level of control over the platform that allows for such changes. Another approach is to create a client-side system that augments the API and provides rollback facilities. An example of such an approach is our previous work on automatic undo for cloud management [1]. This approach works well for small to medium number of consecutive cloud operations (i.e. less than 50). However it does not scale well when the number of operations increases further. The problem in such cases is that the underlying automated planning method from artificial intelligence (AI) faces a computationally hard problem [5].

In this paper, we present a system that facilitates scalable rollback on API controlled could platform using *intermediate checkpoints*. After an initial, user-triggered checkpoint is created, our system intercepts calls to the cloud API and creates intermediate checkpoints as appropriate. Should the need for rollback arise, our system creates a number of AI planning tasks based on the intermediate checkpoints, uses an AI planner to discover appropriate sequences of recovery actions for each planning task, and aggregates them into a complete *rollback plan*. By executing the rollback plan, the cloud resources are brought back to the state of the original checkpoint, thus achieving the goal of the rollback accordingly. To enable the use of AI planning, we rely on an abstract domain model of the API, where each operation is precisely represented with its pre-requisites and effects.

The key challenge this paper addresses is that generating a rollback plan with a large number of steps becomes inefficient at some point. This is due to the inherent complexity of the AI planning problem: AI Planning is known to be PSPACE-complete even in its simplest form [5]. That means that there is no efficient planning algorithm in the general case (unless **P=NP**). The implication for our tool is that, as the number of cloud operations that change the state increases, rollback plan generation time increases exponentially, as confirmed by our experiments.

Our contributions in this paper are the following:

- utilizing a unique feature of the undo scenario, i.e., the possibility to generate intermediate checkpoints, to solve this challenge;
- dividing and parallelizing the rollback plan generation: based on the intermediate checkpoints, we divide the planning task into smaller, independent tasks and parallelize planning process; finally we assemble the resulting partial plans into a comprehensive rollback plan
- evaluating our method by performing rollback for large cloud operations on AWS:

- we show that we can effectively avoid the exponential behavior of naïve planning;
- we also compare plan generation time of our approach on machines with different numbers of CPU cores and show that our approach is highly parallelizable.

The remainder of this paper is organized as follows. The next section discusses scenarios and challenges that motivate our research. We discuss the undo system and different types of checkpoints in Section III. In Section IV, we explain our rollback strategies. In Section V, we analyze the performance of our parallel rollback strategies and compare them with the naïve approach. We discuss related work in Section VI, and finally draw conclusions in Section VII.

II. Background & Motivating Examples

In this section, we discuss some scenarios and challenges of generating undo operations, in part based on our previous work [6]. We also discuss the how previous work has addressed these challenges.

1) *Attaching or Detaching Virtual Disks*
It is possible to invoke a detach or attach operation any time, however doing so could cause failures such as disk inconsistency. A rollback plan needs to either stop the virtual machine, or unmount the volume properly before detach or attach operations. While cloud platforms offer logging service (e.g. AWS CloudTrail [7]), using a log-based approach to execute compensatory operations in reverse order will not suffice in such scenarios.
Insight: Executing compensatory operations in reverse chronological order does not rollback some operations.

2) *Creating and Deleting Clusters*
Auto-scaling mechanisms, like on AWS, can create and maintain a number of virtual machines in a cluster, and automatically supply new machines to replace ones that shut down or fail. In platforms like AWS, creation of cluster cannot be compensated by deletion. Deletion can only be performed when a cluster has no machines. A rollback plan that needs to delete a cluster has to first set the cluster size to zero, and execute delete operation only after all machines have been shut down.
Insight: Compensatory operation (provided by the API) does not always reverse a forward operation.

3) *Creating and Deleting Backups*
Creating backup virtual machines (VMs) requires a sequence of operations such as creating snapshot, creating empty instances and volumes, and copying data. To undo the effects, it suffices to simply delete the backup. Execution of compensating action in reverse order entails executing unnecessary operations (e.g. copying data back).
Insight: The best rollback plan is not necessarily a reverse sequence of compensatory actions.

4) *Deleting a Virtual Disk*
For delete operations, no compensating action may be provided by the API. Although it is possible to recover data stored in virtual disk by using backups, the disk itself cannot be restored once it is deleted. Also, any data added to or changed on the disk after creating a backup is lost irreversibly.
Insight: Some operations provided by APIs are irreversible.

5) *Deploying Application at Scale*
Deployment at scale can be complex and require a large number of API invocations. For example, deployment of new version of a web application may require creating virtual instances and volumes, creating backups, changing auto-scaling configuration, scaling instances up or down, configuring load balancers, changing DNS entries, etc. In such scenarios, the probability of errors is high – especially if the deployment is not automated.
Insight: Large-scale deployments require a large number of API invocations, and have higher probability of failure.

Utilizing an AI planner, we can generate a sequence of rollback actions. Typically, the planner generates the shortest possible plan, i.e., with the least number of actions. This requires a domain model of the API. Analyzing such a model can provide guarantees which actions can be undone, and under which circumstances [6].

Irreversible operations can be facilitated by providing a wrapper around irreversible API operations, where the wrapper replaces the irreversible actions with pseudo variants, e.g., delete is replaced with *pseudo-delete*. In [1], we proposed an approach that wraps AWS's cloud management API and uses a formal domain model to generate undo sequences for rollback. A single AI planning process generates undo sequence based on the domain model and information of the current and a previously checkpointed state of resources. Delete operations are replaced with pseudo-delete: calling a delete operation will only flag the resource as deleted; from that point on, wrapped actions affecting the resource behave as if it actually was deleted. When an administrator requests rollback, the resource is undeleted; in contrast, only when the administrator commits the changes, all flagged resources are actually deleted.

Our previous research did not address improving the efficiency of undo plan generation. In large scale deployment scenarios, the approach fails to scale plan generation time with number of operations. In this paper, we address this issue by extending the previous work.

III. Undo System

In this section, we explain how the undo system discovers and executes a sequence of undo operations, based on the state of cloud resources.

A. Overview and API Wrapper

The system has two main parts: an API wrapper, which intercepts calls to Amazon EC2 API and offers few additional commands, and an AI planner, which generates rollback actions based on checkpointed states.

In a typical usage scenario, an administrator (or script) issues a *checkpoint command* before making changes to resources. The system then gathers relevant information about the state of the cloud resources, and stores this as a checkpoint. After creating checkpoints, administrators can make changes to resources. At this point, the API wrapper transparently replaces irreversible operation with reversible ones and offers rollback operation. If an inadvertent change has been made, the administrator issues *rollback command*. In contrast, if the changes are satisfactory, the administrator issues a *commit command* so that irreversible changes are applied to cloud resources. After the commit command is executed, the checkpoint is deleted and rollback is not offered anymore.

To achieve this functionality, the API wrapper intercepts EC2 commands, selectively delegates them to the API, and handles commands for checkpointing, rollback, commit, and undelete. When a rollback command is issued, it creates a planning problem file, and invokes the AI planner to generate list of rollback actions. The API wrapper collects and translates these actions into a bash script, and executes it.

The API wrapper also facilitates *pseudo-delete* operation. Delete operations on the EC2 API are not reversible. When a delete command is executed, the system sets a *delete flag*, indicating that a resource is deleted. The API wrapper intercepts subsequent calls to deleted resource and indicates that the resource does not exist. When a delete operation needs to be reversed, the system removes this flag.

B. AI Planner

We model rollback operation as an AI planning task and use a recent variant [8] of the FF planner [9] to find rollback actions. The FF planner solves planning problem efficiently by combining hill-climbing with systematic forward search, and using heuristics to prune search space. Given a planning task specified in the planning domain definition language (PDDL) [10], it generates a list of actions that solves the task.

A *planning task* comprises objects, predicates, initial state, goal specification, and a set of possible actions. A planner like FF then finds a sequence from the set of actions that, starting from the initial state, can reach the goal state. The variant of FF that we use considers all paths on which the goal can be reached, even if an action does not yield the most desired outcome [8]. Figure 1 shows an illustration of the AI planner.

Objects, initial state, and goal specification are expressed in a PDDL problem file. The undo system creates a PDDL problem file based on checkpoints whenever a rollback operation is necessary. Given two checkpoints X and Y, created in this order,

1) Objects are all known cloud resources,
2) Initial state is the state of cloud resources specified in Y, the current state, and
3) Goal state is the state of cloud resources specified in X, the original checkpoint.

The PDDL problem file is generated by the undo system to achieve rollback, when this operation is called. Since the aim is

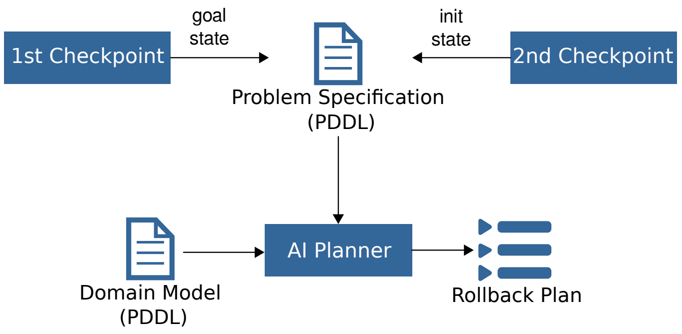

Fig. 1. Using an AI planner to find a rollback plan. The AI planner generates a list of rollback actions based on a problem specification and a domain model file. The problem specification is created using two checkpoints, where the checkpoint created earlier comprises the goal state, and the checkpoint created later comprises the initial state. The domain model is created manually by modeling AWS EC2 API.

```
(define (problem EC2-0)
  (:domain EC2)
  (:objects
    inst00 - tInstance
    vol00 - tVolume
    devName00 - tDeviceName
    AZ0 - tAvailabilityZone)
  (:init
    (instanceRunning inst00)
    (volumeInUse vol00)
    (inAZ vol00 AZ0)
    (inAZ inst00 AZ0)
    (deviceNameInUse devName00 inst00)
    (volumeAttachedToInstance vol00 inst00
        devName00))
  (:goal
    (and
    (volumeAttachedToInstance vol00 inst01
        devName00)
    (instanceRunning inst00))))
```

Listing 1. Sample PDDL problem file

to undo changes, the earlier checkpoint X is the goal, whereas the later checkpoint with the current state of resources is the starting point, the initial state. By reaching the goal, we can effectively revert the changes that have taken place since the original checkpoint was taken. Listing 1 shows an example of a PDDL problem file.

Predicates and Actions are expressed in a PDDL domain model file. In previous work [1], we have manually created a PDDL domain file by modeling AWS EC2 APIs and restricting the domain model to provide guarantees that actions are undoable. If a formal model of the API or the system behind it is available, a PDDL domain file may be generated automatically. In [8], this was done for over 2,000 services offered by SAP systems, using software engineering models from the development of those systems. Listing 2 shows a snippet of PDDL domain file.

In the variant of PDDL we employ, new resources cannot be created out of nowhere. In the domain model, we use the unary predicate *notYetCreated* to enable creation of resources.

```
(define (domain EC2)
  (:types tAMI tInstance ...)
  (:predicates
    (hasAMI ?x0 - tInstance ?x1 - tAMI)
       ... )
  (:constants ...)
  (:action Run-Instance
    :parameters (?ami - tAMI ?inst -
       tInstance ?secGroup -
       tSecurityGroup)
    :precondition
      (and
        (notYetCreated ?inst)
        (activeSecurityGroup ?secGroup))
    :effect
      (oneof
        (and (instanceRunning ?inst)
          (not (instanceStopped ?inst))
          (not (notYetCreated ?inst))
          (belongsToSecurityGroup ?inst ?
             secGroup)
          (hasAMI ?inst ?ami))
        (unrecoverableError ?inst))))
```

Listing 2. Snippet of PDDL domain file

Additionally, the unary predicate *deleted* is used to indicate the deletion of a resource. In relation to that, some minor post-processing of initial and goal states is necessary: objects that were *notYetCreated* in the original checkpoint, but have been created by the time rollback is called, need to be marked as *deleted* in the goal state since they cannot be "un-created". While these two states (not yet created and deleted) are very similar – the respective object is not there – it is important to distinguish them in our models: once a resource with a specific ID has actually been deleted, for most resource types on AWS it is impossible to get exactly this resource back. For resource types like static IP addresses, this distinction can be very important.

C. Additional Operations

Our undo system offers four additional operations: checkpoint, rollback, commit, and undelete.

When a system administrator creates a checkpoint, the system captures the state and relationships of cloud resources. We use different kinds of checkpoints to speed up rollback plan generation time. Checkpoints are explained in the next section in detail.

When a rollback command is issued, the system executes rollback operations based on the checkpoints. AI planning processes find a list of compensating actions, which is converted to a script, and executed.

Commit is a special command that finalizes delete operations. After an initial checkpoint has been taken, we replace delete operations with pseudo-delete, where resources are not actually deleted when the respective operation is called. Only when a Commit command is invoked, the actual delete operation is executed for all pseudo-deleted resources. In addition, any checkpoints taken up to calling commit are removed.

Finally, when delete operations are replaced with pseudo-delete, it becomes possible to undelete resources. To do so, it suffices to remove the "deleted" flag from the resource for which undelete is called. Undelete is also available to the AI planner, as the only operation in the PDDL domain model that is not implemented by AWS but the API wrapper.

D. Checkpoints

A checkpoint is a reference that identifies state of resources on the cloud at a point in time. It captures the state information of all cloud resources, such as which VMs existed, which volumes existed, which VMs were connected to which volumes, etc.

When an administrator wants to have the ability to rollback, she creates a checkpoint before making changes to resources on the cloud. The undo system then gathers information about state and relationships of cloud resources, and saves it to persistent storage. Additionally, the system creates other types of checkpoints depending on the number and type of commands that are called between taking a checkpoint and calling commit or rollback. If called, the rollback operation uses these checkpoints to generate a rollback plan, which is translated into an executable script.

The system uses three types of checkpoints: (i) manual checkpoint, (ii) intermediate checkpoint, and (iii) current checkpoint. All three types of checkpoints contain the same kind of information about cloud resources and their states, for use by the AI planner when generating rollback plans.

1) Manual Checkpoint: a manual checkpoint represents a consistent state to which a system can be rolled back. This is a checkpoint created by the system administrator manually before making changes to resources on cloud. The manual checkpoint information is stored persistently.

2) Intermediate Checkpoint: This is a checkpoint created by the undo system after a certain number of *change commands*, i.e., commands which change state of the system, have been executed. Intermediate checkpoints are stored persistently. They are used to improve rollback plan generation time.

3) Current Checkpoint: This is a temporary checkpoint object created by the system when rollback command is triggered. The current checkpoint does not need to be stored persistently, as it is used only when the rollback command is executed.

Intermediate Checkpoint plays a significant role in improving the scalability of rollback plan generation. They allow us to divide the planning process into smaller independent tasks, where each task is responsible for generating rollback actions between two adjacent checkpoints. Overall plan generation time can be reduced significantly when these tasks are parallelized.

The API wrapper tracks the number of called change commands, and creates an intermediate checkpoint after every *n-th* change command. The number of commands, *n*, after which an intermediate checkpoint is created is configurable. All details about the creation of intermediate checkpoints are hidden from the user.

One of the challenges in creating the checkpoints is that, when cloud system is very large and there are too many cloud resources to be captured, checkpoint creation time is relatively long. Therefore, choosing the number of change commands for intermediate checkpoints, n, represents a trade-off between user experience and plan generation time.

IV. Rollback Strategies

Rollback is the core function of our undo system. The rollback command returns the resources to their state at the point in time when a *manual checkpoint* was created. The undo system uses the AI planner to find a rollback plan of undo actions, generates a bash script based on the rollback plan, and executes the script.

Our approach comprises three strategies to perform rollback, which we describe below. They differ in how they use the various checkpoints to generate a rollback plan. In the next section, we comparatively evaluate the approaches and other aspects.

A. Manual Checkpoint Rollback (MCR)

This rollback strategy is a naive approach where the rollback command creates a PDDL problem file from solely the *current checkpoint* and the *manual checkpoint*. The state captured in the current checkpoint is used as initial state, and the state captured in the manual checkpoint is used as goal state.

The drawback of this approach is that it does not consider plan length (number of actions in the undo plan). Because planning is PSPACE-hard, the runtime of undo plan generation using this approach will increase exponentially over increasing plan length. The MCR strategy is the base case from our prior work [1], and the goal of the other two strategies is to improve over MCR when creating long rollback plans. To improve plan generation time for large plans, they make use of *intermediate checkpoints*.

B. Intermediate Checkpoint Rollback (ICR)

In this rollback strategy, user commands are tracked to create *intermediate checkpoints*, as described in Section III-D. As stated earlier, this strategy is unique to the rollback scenario: only because we can observe the “do”-actions and can create intermediate checkpoints, we can make use of them for efficient rollback.

When the rollback command is invoked for ICR, the undo system reads each pair of adjacent checkpoints in reverse chronological order of their creation, and generates appropriate problem PDDL files. This set of PDDL problem files is used by the planner to generate a set of sequences of rollback actions.

For example, say there are 3 intermediate checkpoints C_1, C_2, and C_3, as well as the current checkpoint, CC, and the manual checkpoint, MC. Then ICR creates four checkpoint pairs: $(CC, C_3), (C_3, C_2), (C_2, C_1)$, and (C_1, MC). For every checkpoint pair (X, Y), the system generates a planning task with X as initial state and Y as goal state.

Rollback plans for each checkpoint pair are generated in parallel. Once all partial rollback plans are available, they are

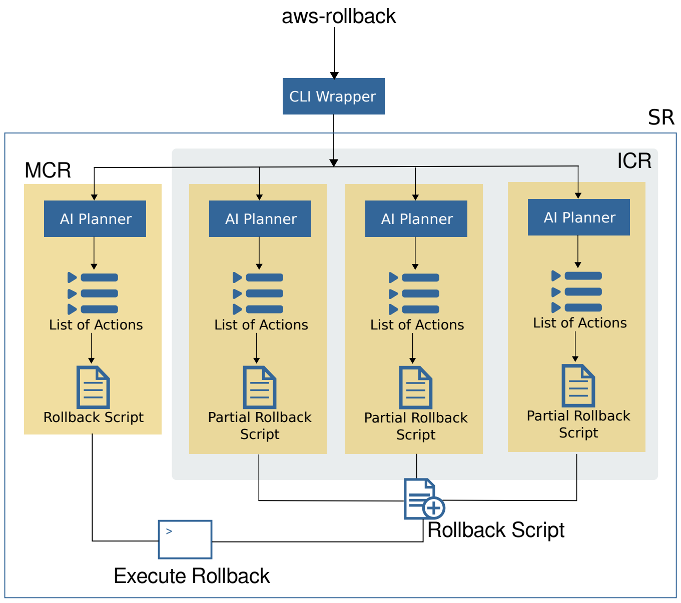

Fig. 2. Overview of all three rollback strategies: MCR, ICR and SR. After the rollback command is issued, the API Wrapper creates a problem specification and invokes the planner for only MCR, or in several times in parallel for ICR, or both for SR. Partial rollback scripts generated by ICR are concatenated into a single script. For SR, once a rollback plan is generated by either ICR or MCR, the respective other planning processes are stopped and the rollback plan is executed.

concatenated to form a single sequence of commands, the end-to-end rollback plan, which is then translated and executed as before.

C. Scalable Rollback (SR)

Scalable Rollback is a rollback strategy that combines MCR and ICR. Figure 2 shows how all three strategies work, and how SR combines the other two approaches.

SR runs MCR and ICR in parallel when the rollback command is called. ICR, in turn, parallelises its planning process into the generation of partial rollback scripts, and concatenates them to a single script. The system executes whichever rollback script is generated earliest. All parallel computation is terminated once either MCR or ICR has generated a rollback script. As such, SR performs a race between MCR and ICR.

V. Evaluation

In this section we describe the experiments we conducted to evaluate the performance of the three rollback strategies, and discuss results and threats to validity. To compare the three strategies from the previous section, we focus on plan generation times when varying (i) plan length and (ii) the number of cores available for planning.

A. Experiment Setup

In our test scenarios, configuration of a collection of instances and volumes was modified by detaching and attaching volumes to different instances. In our experiments, the number of cloud resources forming the states is the same for all plan lengths, so as to keep this factor stable. We created scenarios where an optimal rollback plan has a plan length of $m \in \{16, 32, 48, 64, 80, 96, 112, 128, 144\}$. The scenarios were set up so that the rollback plans generated by all approaches had same number of steps. This was done to ensure that the planning tasks were comparable over all three strategies.

For collecting the checkpoints, we ran scripts that started with creating a manual checkpoint, executed the "do" actions on AWS that were necessary to reach the state from which we wanted to roll back while the undo system collected intermediate checkpoints, and saved the final state as current checkpoint. Therefore, the checkpoints were all based on observation of real states of AWS resources. We then used the collected checkpoints, but without always executing the resulting rollback plans, to save time and cost.

Intermediate checkpoints were created with $n = 20$, i.e., after every 20th command that changes the state of the system. As stated earlier, the number of commands n after which intermediate checkpoint is created presents a trade-off in user experience: on the one hand creating a checkpoint takes time, therefore doing so less often (larger n) is preferable. On the other hand, if n is too large, then rollback cannot profit much from ICR or SR, and planning for rollback will take very long, so a smaller n is preferable. In our usage scenarios, we found that $n = 20$ was a good compromise.

Using the checkpoint sets collected as per above, we ran plan generation time for each plan length and rollback strategy for 25 iterations. We measured the time for plan generation and report average values over the 25 iterations here. Since standard deviation and standard error of mean were low for the values from the 25 iterations, this number was deemed sufficient.

The experiments used AWS EC2 machines for plan generation, namely C4.xlarge instances with 4.5 GB memory and 4 vCPUs, C4.2xlarge instances with 15 GB memory and 8 vCPUs, and C4.4xlarge instances with 30 GB memory and 16 vCPUs. It should be noted that a vCPU is a hyperthread on a CPU core of an Intel Xeon processor.[1] Therefore, 4 vCPUs correspond to two physical cores, for instance.

B. Plan Generation Time vs. Rollback Strategy

Using the above setup, we compared the rollback plan generation time for different plan lengths and the three strategies: Manual Checkpoint Rollback (MCR), Intermediate Checkpoint Rollback (ICR), and Scalable Rollback (SR). The results of this experiment is shown in Figure 3.

First we observe that the runtime of MCR indeed exhibits exponential behavior: the respective curve is roughly a straight line, but on a logarithmic scale. In contrast, the ICR and SR strategies scale up almost linearly. As plan length increases, ICR and SR offer more and more improvement over MCR in terms of plan generation time.

While the ICR strategy uses all available vCPUs to generate plans using intermediate checkpoints, SR allocates one vCPU for generating rollback plan using MCR approach, and the rest for ICR approach. Therefore the ICR approach finds plans more quickly. Due to the fact that a vCPU is only a

[1]http://aws.amazon.com/ec2/instance-types/, accessed on 14/6/2015.

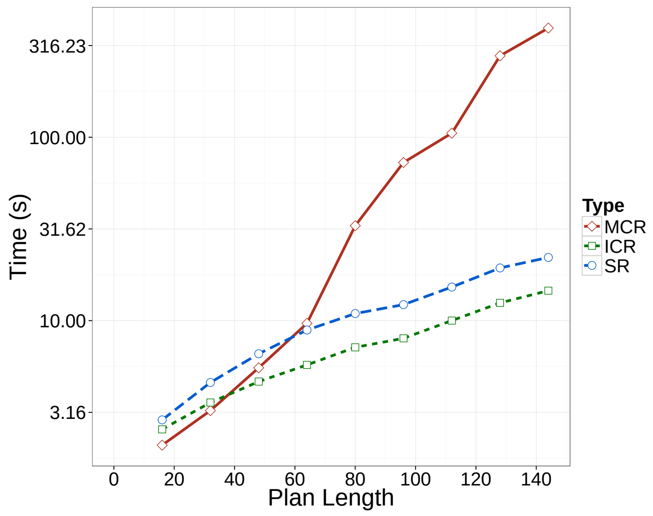

Fig. 3. Runtime of plan generation (in seconds) using the strategies MCR, ICR, and SR. Note that the y-axis uses a *logarithmic scale*. Plan generation time was measured on a VM with 4 virtual CPU cores (C4.xlarge).

hyperthread, not a physical core, it is hard to hypothesize on how the MCR computation influences the parallel threads. From the data, it appears that ICR takes typically 20-30% less time than SR, once an intermediate checkpoint has been created.

C. Scalability

To measure how well SR can scale when given more compute power, we ran the experiments where we varied plan length and the machine type. In particular, we used three different types of AWS C4 instances: C4.xlarge, C4.2xlarge, and C4.4xlarge. The numbers for vCPUs and memory can be found in Section V-A. The results of this experiment are shown in Figure 4.

We observe that plan generation time improves strongly when the number of vCPUs is increased from 4 to 8. However, when it is increased from 8 to 16 vCPUs, we only see minor improvement. While additional CPU cores can parallelize the planning processes, the centralized overhead (reading files, generating planning problems, managing threads, and aggregating the results) attribute for an increasing runtime for rollback plan generation. Our undo system prototype is not fully optimized, so we believe more performance gains can be realized by careful performance tuning.

Two points should be noted. First, while both of our parallelizing strategies SR and ICR can benefit from additional cores, MCR cannot. Second, the approach of using intermediate checkpoints is not generally applicable, but specifically relies on the undo scenario where "do"-actions can be monitored and intermediate checkpointing can be realized.

D. Discussion

The results shown here are highly encouraging with respect to the direction of parallelizing the planning task using intermediate checkpoints. Comparing the most extreme case, i.e., rollback plan of length 144, MCR takes 394.08 seconds

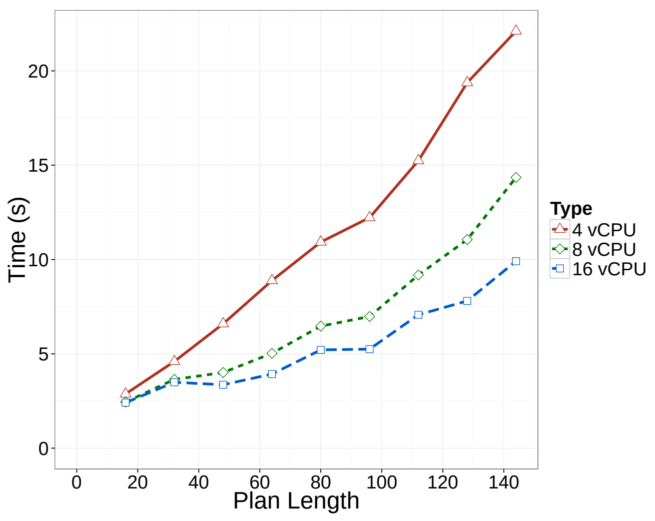

Fig. 4. Runtime of the SR strategy, on a *linear scale* using VMs with a varying number of vCPUs.

as opposed to 9.91 seconds for SR on a 16vCPU machine. While the comparison of MCR on a 4vCPU machine with SR on a 16vCPU machine has to be taken with a grain of salt – see the discussion below – this corresponds to a *speedup of factor 39.76*. By circumventing the need for solving the whole end-to-end planning task, we avoid the exponential behavior of AI planning and achieve roughly linear scalability. There are, however, limitations and threats to validity in our study, which we will discuss next.

One of the limitations of our approach is that it does not consider the quality of rollback plans. In the experiments here, this has no effect since all strategies lead to the same plans as per our experiment design. In general, though, rollback plans generated using *Intermediate Checkpoint Rollback* are not guaranteed to be optimal in terms of plan length. For example, consider a case like Scenario 3) in Section II, where a system administrator executes a large number of commands and reaches a state which can be rolled back by a small number of steps. Rollback plan generation using the ICR approach will track all commands that change the state of the system, consider all intermediate checkpoints, and likely generate a suboptimal plan. In contrast, the MCR strategy typically generates an optimal plan. By using the SR strategy, an optimal plan is generated only if the MCR planning process completes first. In our future work, we will consider alternative methods that take plan quality into account. This is a tradeoff that may require user input: how much longer are you willing to wait for a better plan?

In this work there are few threats to validity:

1) Internal Validity: In our experiments, the memory size is larger for machines with a larger number of CPU cores. Plan generation time may or may not be affected by this factor – the planner might not use the additional memory. We are, as yet, not certain regarding the extent of the effect. Also, our experiments were run on virtual machines beyond our direct control. Computation on resources that are potentially shared with other users of the cloud may be subject to significant performance variation, and even machines of the same type are known to have fluctuating performance [11].

2) External Validity: In our experiment scenarios, we performed a subset of available actions (attach, detach, start, and stop). We cannot guarantee that plan generation times will be similar when a different subset of actions are performed. Furthermore, a user's network connection to the internet may introduce latencies and outages that we did not account for in our studies.

VI. Related Work

Rollback methods can be checkpoint-based [12], [13], log-based [12], [14], or shadow page-based [14]. Our approach is checkpoint-based, where checkpoints store relevant information on the disk. In our approach, rollback means achieving physical state of cloud resources that match the state stored in checkpoint. In contrast to other checkpoint based approaches, where rollback is preformed by copying saved information back to memory, our rollback operation requires execution of API operations. This is similar to *Sagas* [15], where system designers provide compensating actions for each operation, so that undo operations would require execution of corresponding compensation actions in reverse order. However, as argued in Sections I and II, undoing actions in reverse chronological order can be highly suboptimal.

AI planning has been used for system configuration, e.g. [16]–[22], and cloud configuration, e.g. [23]–[25]. In [24] planning is applied in a straight-forward fashion to the problem of reaching user-specified goal. The work is integrated into cloud management tools such as Facter, Puppet, and ControlTier. In [25] hierarchial task network (HTN) planning is applied at PaaS level for fault recovery. Similarly, [23] uses HTN planning to achieve configuration changes for system modeled in common interface model (CIM) standard. Besides AI planning, POMPDPs (Partially Oservable Markov Decision Processes) can be used for recovery process [26]. Using intermediate checkpoints to speed up planning for undo has not been the subject of any works we are aware of.

In [8], so-called Status and Action Management (SAM) models which describes Business Object behaviors are automatically translated to PDDL domain models. Since a formal model comparable to SAM is not available for AWS EC2 [4], we have manually modeled AWS EC2 API in PDDL form. In [1], AI planning is used to generate rollback plans on cloud operations. Our work extends this approach by dividing and parallelizing planning processes.

VII. Conclusion

To reduce the impact of human-induced errors in API-controlled cloud environments, it can be highly beneficial to provide rollback of inadvertent changes made by an administrator. Our prior approach works well for rollback with a small number of actions, but does not scale as the number of actions increases.

In this paper, we present a scalable approach to rolling back cloud operations using AI planning. Our undo system intercepts API calls, creates checkpoints, uses an AI planner to generate a plan that takes the system back to previous state, and finally executes the plan.

We discuss three strategies to generate rollback plans. Through experiments we demonstrate that, compared to the naïve approach, our Scalable Rollback approach improves plan generation time as plan length increases. For the most extreme case we tested, we observe speedup by a factor of nearly 40.

In future work, we plan to consider the quality of generated plans, as well as further to optimize plan generation time by filtering and reducing the state space the AI planner has to consider.

Acknowledgment

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] I. Weber, H. Wada, A. Fekete, A. Liu, and L. Bass, "Automatic undo for cloud management via ai planning," in *Proceedings Usenix Workshop on Hot Topics in System Dependability*, 2012.

[2] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, ser. USITS'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251460.1251461

[3] J. Gray, "Why do computers stop and what can be done about it?" in *Symposium on Reliability in Distributed Software and Database Systems*. IEEE Computer Society, 1986, pp. 3–12. [Online]. Available: http://dblp.uni-trier.de/db/conf/srds/srds86.html

[4] "AWS Elastic Compute Cloud (EC2)," http://aws.amazon.com/ec2/, accessed: 2015-06-02.

[5] T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, pp. 165–204, 1994.

[6] I. Weber, H. Wada, A. Fekete, A. Liu, and L. Bass, "Supporting undoability in systems operations," in *USENIX LISA'13: Large Installation System Administration Conference*, Washington, DC, USA, Nov. 2013.

[7] "AWS CloudTrail," http://aws.amazon.com/cloudtrail/, accessed: 2015-06-02.

[8] J. Hoffmann, I. Weber, and F. M. Kraft, "SAP speaks PDDL: Exploiting a software-engineering model for planning in business process management," *Journal of Artificial Intelligence Research (JAIR)*, vol. 44, pp. 587–632, 2012.

[9] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *J. Artif. Int. Res.*, vol. 14, no. 1, pp. 253–302, May 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=1622394.1622404

[10] D. McDermott *et al.*, *The PDDL Planning Domain Definition Language*, The AIPS-98 Planning Competition Comitee, 1998.

[11] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[12] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep. 2002. [Online]. Available: http://doi.acm.org/10.1145/568522.568525

[13] B. Randell, "System structure for software fault tolerance," *IEEE Transactions On Software Engineering*, vol. 1, no. 2, 1975.

[14] G. Graefe, "A survey of b-tree logging and recovery techniques," *ACM Trans. Database Syst.*, vol. 37, no. 1, pp. 1:1–1:35, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2109196.2109197

[15] H. Garcia-Molina, "K. salem. sagas," in *Proceedings of the 1987 ACM SIGMOD Conference*, 1987.

[16] N. Arshad, D. Heimbigner, and A. L. Wolf, "Deployment and dynamic reconfiguration planning for distributed software systems," in *Tools with Artificial Intelligence, 2003. Proceedings. 15th IEEE International Conference on*. IEEE, 2003, pp. 39–46.

[17] N. Arshad, D. Heimbigner, and A. Wolf, "A planning based approach to failure recovery in distributed systems," in *WOSS'04: 1st ACM SIGSOFT Workshop on Self-Managed Systems*, 2004, pp. 8–12.

[18] A. J. Coles, A. I. Coles, and S. Gilmore, "Configuring service-oriented systems using PEPA and AI planning," in *Proceedings of the 8th Workshop on Process Algebra and Stochastically Timed Activities (PASTA 2009)*, August 2009.

[19] C. E. da Silva and R. de Lemos, "A framework for automatic generation of processes for self-adaptive software systems," *Informatica*, vol. 35, pp. 3–13, 2011, publisher: Slovenian Society Informatika.

[20] F. Dalpiaz, P. Giorgini, and J. Mylopoulos, "Adaptive socio-technical systems: a requirements-driven approach." *Requirements Engineering*, 2012, springer, to appear.

[21] B. Drabble, J. Dalton, and A. Tate, "Repairing plans on-the-fly," in *Proc. of the NASA Workshop on Planning and Scheduling for Space*, 1997.

[22] K. Levanti and A. Ranganathan, "Planning-based configuration and management of distributed systems," in *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, ser. IM'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 65–72. [Online]. Available: http://dl.acm.org/citation.cfm?id=1688933.1688942

[23] S. Hagen and A. Kemper, "Model-based planning for state-related changes to infrastructure and software as a service instances in large data centers," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, ser. CLOUD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 11–18. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2010.14

[24] H. Herry, P. Anderson, and G. Wickler, "Automated planning for configuration changes," in *LISA'11: Large Installation System Administration Conference*, 2011.

[25] F. Liu, V. Danciu, and P. Kerestey, "A framework for automated fault recovery planning in large-scale virtualized infrastructures," in *MACE 2010, LNCS 6473*, 2010, pp. 113–123.

[26] K. R. Joshi, W. H. Sanders, M. A. Hiltunen, and R. D. Schlichting, "Automatic model-driven recovery in distributed systems," in *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, ser. SRDS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 25–38. [Online]. Available: http://dx.doi.org/10.1109/RELDIS.2005.11