Environment-Sensitive Performance Tuning for Distributed Service Orchestration

Yu Lin[1], Franjo Ivančić*[2], Pallavi Joshi[2], Gogul Balakrishnan[2], Malay Ganai[2], and Aarti Gupta[2]

[1] University of Illinois at Urbana-Champaign, Champaign IL, USA
[2] NEC Laboratories America, Princeton, NJ, USA

Abstract. Modern distributed systems are designed to tolerate unreliable environments, i.e., they aim to provide services even when some failures happen in the underlying hardware or network. However, the impact of unreliable environments can be significant on the performance of the distributed systems, which should be considered when deploying the services. In this paper, we present an approach to optimize the performance of distributed systems under unreliable deployed environments, through searching for optimal configuration parameters. To simulate an unreliable environment, we inject several failures in the environment of a service application, such as node crash in the cluster, network failures between nodes, resource contention in nodes, etc. Then, we use a search algorithm to find the optimal parameters automatically in the user-selected parameter space, under the unreliable environment we created. We have implemented our approach in a testing-based framework and applied it to several well-known distributed service systems.

1 Introduction

Effectively tuning and optimizing performance of distributed services is essential for controlling costs and improving customer satisfaction. Distributed applications often provide configuration parameters that can be tuned by system administrators for a specific system deployment. However, finding the best configuration for distributed systems is challenging. First, the parameter space of a distributed service can be large, and it is hard to find the best configuration manually. For example, Hadoop contains more than 190 parameters that are specified to control the behavior of a MapReduce job. Second, the best configuration can be different under different deploying environments, or even for different workloads. Third, the best configuration can also be affected by anomalies in the deployed environment, such as in the network (link failures, link congestions, packet drops), node (process crash, memory contention, deadlocks, CPU overload), or disk (slow response, failures, corruption). These anomalies can impact both the correctness and performance of distributed services.

* Current affiliation: Google, Inc.

Researchers have proposed several ways to mitigate the above challenges. Some approaches [6] are proposed to search the optimal configuration automatically for a given workload in large parameter spaces. However, these approaches only target some specific distributed applications/systems and do not consider the possible effects of anomalies in the environment (i.e., they assume an ideal environment without any failures). Some perturbation-based testing approaches [10] are also proposed to test the robustness or availability of the distributed services under environmental anomalies, but they do not consider performance.

This paper proposes an approach to optimize the performance of distributed services in the presence of environmental anomalies, such as node VM crash, network failure, etc. Our approach injects *disturbance actions*, which simulate the environmental anomalies, into the environment. Examples of disturbance actions include shutting down a node in a cluster or in a cloud, disconnecting the link between nodes, limiting the resources a node can use, etc. Then, we use a search algorithm to find the optimal configuration for a distributed system that is deployed in the disturbed environment we created. Since various disturbed environments can be created by applying different combinations of disturbance actions, and it is impractical to optimize performance over all combinations, we also propose two strategies to select the disturbed environments.

This paper makes the following contributions:

- We formalize the problem of finding good configuration parameters for performance optimization of distributed services in the presence of environmental anomalies such as network failures or delays, node crashes, and multi-tenant resource contentions.
- We have implemented our approach for arbitrary distributed systems in a framework called RIOT, which is based on an automatic service orchestration framework Juju [2]. We have extended Juju to allow deployment-time queries on performance and health of the services, and combine automatic service deployment with anomaly injections.
- We present initial promising experiments on some popular distributed applications, including Apache `Hadoop`, `HBase` and `ZooKeeper`.

2 Problem Definition

We consider software performance to include the response time and system resource (e.g., CPU/memory) consumption [9]. We assume that the performance of a distributed application depends on the deployed environment (e.g., virtualized or cloud), the workload it runs, and the configuration. Thus, the performance can be denoted by formula 1, in which P is the performance, E is the environment, Ψ is the workload, and $\boldsymbol{c}$ is the configuration. P is denoted as a function $\mathcal{F}$ of E, Ψ and $\boldsymbol{c}$.

$$P = \mathcal{F}(E, \Psi, \boldsymbol{c}) \quad (1)$$

In our approach, we inject a set of disturbance actions in the environment to simulate the environmental anomalies. Thus, the environment E can be denoted

by a function Γ over a set $\boldsymbol{a}$ that contains disturbance actions (formula 2). Note that the ideal environment is when $\boldsymbol{a} = \emptyset$.

$$E = \Gamma(\boldsymbol{a}), \text{where } \boldsymbol{a} = \{a_1, a_2, ..., a_n\} \tag{2}$$

For a given workload Ψ, our objective is to find the optimal configuration $\boldsymbol{c}_{opt}$ which leads to the best performance in the parameter space S, under an environment E (formula 3).

$$\boldsymbol{c}_{opt} = \underset{\boldsymbol{c} \in S}{\operatorname{argmin}} \digamma(E, \Psi, \boldsymbol{c}), \text{where } \boldsymbol{c} = \{c_1, c_2, ...\} \tag{3}$$

However, for a disturbance action set of size n, we may need to consider 2^n environments, taking into account all possible combinations. If we further allow multiple occurrences of each disturbance action, then the combinatoric problem becomes even worse. It is not practical to find the best configuration for each possible environment. Thus, we propose two strategies to select the environment to consider.

For a given action set $\boldsymbol{a}$, let $\boldsymbol{A}$ denote a set of disturbance sets whose elements are subsets of $\boldsymbol{a}$ (formula 4). $\boldsymbol{A}$ can be selected by a user, based on his knowledge of the actual deployed environment.

$$\boldsymbol{A} = \{A_1, A_2, ..., A_m\}, \text{where } A_i \text{ is a subset of } \boldsymbol{a} \tag{4}$$

Based on the selected $\boldsymbol{A}$, we can apply the following strategies:

1) Min-max game strategy: Optimize configuration for the disturbance set A_{worst} which leads to the worst performance ($A_{worst} \in \boldsymbol{A}$). In this strategy, the performance can be denoted as $P = \digamma(\Gamma(A_{worst}), \Psi, \boldsymbol{c})$. We use this strategy in our experiment, where we set $\boldsymbol{A}$ to $\{\{a_1\}, \{a_2\}, ..., \{a_n\}\}$.

2) Weighted average strategy: Optimize configuration for expected performance using probabilistic estimations about the frequency of disturbances. In this strategy, the performance can be denoted as $P = \sum_{i=1}^{m} w_i * \digamma(\Gamma(A_i), \Psi, \boldsymbol{c})$. The weight w_i can be viewed as the estimated probability that a disturbance set may occur in the deployed environment.

3 Environment-Sensitive Performance Tuning Framework

Our approach relies on service orchestration frameworks, which automate the process of deploying, configuring and maintaining the distributed services efficiently. We propose new service orchestration commands that enable us to query performance or correctness of the underlying deployed services. We also integrate the orchestration framework with disturbance action injection during test execution. In this paper, we use Juju [2] as the service orchestration framework. Juju uses Charms [3] to deploy the application and supports many popular cloud environments, such as EC2 and OpenStack. Charms essentially comprise scripts which define the ways of deploying and managing the distributed services.

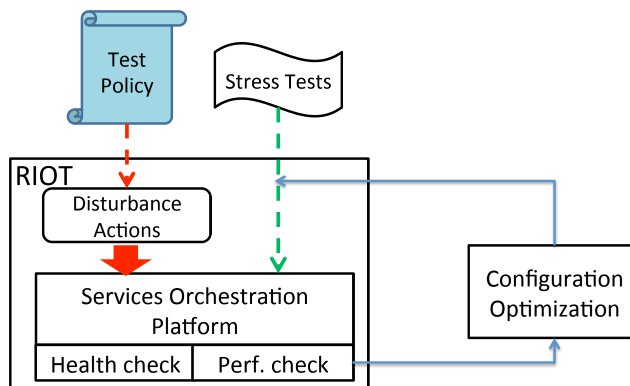

Fig. 1. An overview of RIOT.

Figure 1 shows an overview of RIOT, our framework for testing and tuning performance of distributed services. In addition to deploying distributed services, we create a test policy which defines a set of disturbance actions and the frequency of applying them. After applying the test policy, RIOT runs workloads (i.e., stress tests) under the disturbed environment, and queries the performance through the service orchestration commands (*performance check* in Fig. 1). The information to query is defined in Juju Charms. For example, in our experiments, we execute a script to query performance counters from Ganglia [1] in the Charms. Note that we also propose *health checks* for checking correctness, availability, or other requirements of interest. However, we target performance in this paper. After obtaining the performance in one execution, RIOT reconfigures the services dynamically through Juju to search for a better configuration, and the above process is repeated until some time limit.

3.1 Disturbance Actions

As a proof-of-concept, we have implemented the following disturbance actions in RIOT:

- *Shutting down a node*, which simulates a node crash in a VM/cluster/cloud.
- *Disconnecting two nodes*, which simulates a link failure.
- *Limiting the available memory in a node*, which simulates resource contention in a node.

Juju creates Linux LXC containers [4] and deploys the service units in the containers (i.e., the service processes are created by the containers). We shut down a node by stopping the corresponding LXC container. To disconnect two nodes, we use Linux traffic control to drop all packets sent between the two containers. We use LXC `cgroup` to limit the available memory, since this command allows specifying resource limits for a control group of processes. It is easy to define other disturbance actions, such as link congestions, packet delay using Linux traffic control tools, for example.

Table 1. Ganglia performance counters and weights used in the experiment.

Name	Description	Weight
Bytes in	Number of bytes in per second	0.1
Bytes out	Number of bytes out per second	0.1
CPU user	Percent of user CPU	0.2
Used memory	Amount of used memory	0.1
Execution time	Execution time of the stress tests	0.5

3.2 Searching Configuration Parameters

As mentioned earlier, the search algorithm executes the stress tests repeatedly within a given time limit to search for the optimal configuration. For efficiency and effectiveness, the algorithm must minimize the number of stress test executions while finding near-optimal configurations. For this purpose, RIOT uses the *Recursive Random Search (RRS)* algorithm [12], which has been applied to black-box optimization problems. RRS first samples the search space randomly for a certain number of times, to identify regions that contain optimal solutions with high probability. These regions are then sampled recursively, and are either moved or shrunk gradually to local optima based on the samples. Then, RRS restarts random sampling to find better regions based on the result of the last iteration, and repeats the recursive search.

3.3 Performance Measurement

To compare the performance of different stress test executions, we have to measure and quantify the performance of an execution. We use Ganglia [1] as our performance measurement tool. Ganglia is a scalable distributed monitoring system for high-performance computing systems, including clouds. It provides around 50 different performance counters. RIOT deploys Ganglia along with the distributed services. During the execution of stress tests, RIOT queries the performance counters from Ganglia (i.e., the *performance check* module in Fig. 1) every 30 seconds, and calculates the average for each performance counter. The weighted average of these performance counter values is then used to represent the performance of one execution. A larger value means worse performance. Table 1 shows the performance counters and weights used in our experiments. Note that the weights we used are heuristic , and users can select other weights based on their estimated importance for each performance counter.

4 Experiments

Experimental Setup. We applied RIOT on three well-known distributed applications in our experiments: `Hadoop`, `HBase` , and `ZooKeeper`. `Hadoop` is a framework for distributed processing of large data sets using map-reduce programming models, while `HBase` is a distributed and scalable Big Data database for `Hadoop`. `ZooKeeper` is a centralized service for providing distributed synchronization. (We used versions 1.0.2, 0.92.1, and 3.3.5, respectively.) The experiments are run in a

Table 2. Configuration parameters used in the experiments.

Application	#	Parameter	Values
`Hadoop`	c_1	number_of_datanode	3, 4, 5, 6, 7
	c_2	number_of_tasktracker	3, 4, 5, 6, 7
	c_3	io_sort_mb	80, 100, 120, 140, 160, 180, 200
	c_4	io_sort_factor	8, 9, 10, 11,12
`Hadoop` & `HBase`	c_5	dfs_block_size (MB)	32, 64, 128, 256
	c_6	datanode_max_xcievers	2048, 4096, 6144
	c_7	namenode_handler_count	8, 9, 10, 11,12
	c_8	heap (MB)	512, 1024, 1536, 2048
`ZooKeeper`	c_9	number_of_zookeeper_node	3, 4, 5, 6, 7
	c_{10}	default_group	0, 1, 2, 3, 4
	c_{11}	default_weight	1, 2, 3, 4, 5

cloud where each node has a 2.33GHz 4-cores Intel Xeon processor, 2GB memory and runs Ubuntu 12.04.1. The time limit for RRS is set to two hours.

The configuration parameters used in the experiments are shown in Table 2. Column 1 shows the application, and Column 2 the parameter number (for ease of reference). Columns 3 and 4 show parameter names and values (i.e., parameter space), respectively. Note that we consider the number of nodes also as configuration parameters (e.g., `number_of_datanode` and `number_of_tasktracker`).

Table 3. Workloads and Results of Experiments.

Stress Test	a_{worst}	Performance Value			Optimal Configuration	
		E_i	E_d^{ic}	E_d	E_i	E_d
TestDFSIO	disconnecting two `datanodes`	1	1.22	1.15	c_1=5, c_3=100, c_4= 10, c_5=64, c_6=4096, c_8=2048	c_1=6, c_3=80, c_4=12, c_5=32, c_6=4096, c_8=2048
TeraSort	disconnecting two `tasktrackers`	1	1.20	1.06	c_2=3, c_3=100, c_4=10, c_5=32, c_8=2048	c_2=6, c_3=80, c_4=10, c_5=32, c_8=2048
NNBench	limiting memory of a `tasktracker`	1	1.80	1.05	c_2=3, c_3=120, c_4=10, c_5=256, c_8=1536	c_2=6, c_3=160, c_4=12, c_5=256, c_8=2048
MRBench	limiting memory of a `tasktracker`	1	1.30	1.12	c_2=3, c_3=100, c_4=10, c_5=64, c_8=1024	c_2=4, c_3=160, c_4=10, c_5=64, c_8=1536
HBase	disconnecting a `RegionServer` with `Master`	1	1.53	1.06	c_5 = 64, c_6=4096, c_7=10, c_8=1024	c_5 = 32, c_6=4096, c_7=8, c_8=1536
ZooKeeper	shutting down a node	1	1.07	1.07	c_9 = 5, c_{10}=0, c_{11}=2	c_9 = 7, c_{10}=4, c_{11}=1

RIOT runs workloads (i.e., stress tests) on the applications to test the performance. For `Hadoop`, we used four stress tests, shown in the first column of Table 3. These tests cover different layers of `Hadoop`: `TestDFSIO` is a read and write test for `HDFS`, `TeraSort` is a large-scale test that covers both map-reduce and `HDFS` layers, `NNBench` tests the `NameNode` hardware and configuration, `MRBench` focuses on the map-reduce layer by running many small map-reduce jobs. For `HBase`, we used a stand-alone `HBase` installation with one master server and three region servers, without integrating it with `Hadoop`. We randomly generated `HBase` operations as workload, including adding/deleting tables, adding/deleting tuples, and queries. Similar to `HBase`, we randomly generated workloads for `ZooKeeper`,

including stopping and re-starting the ZooKeeper service, creating new elements in the shared configuration state, re-setting values, querying some states, or deleting some states. For each application, we injected three disturbance actions (described in Sec. 3.1), with one action per execution (i.e., we did not combine disturbance actions).

Results. The workloads and experimental results are shown in Table 3. Workloads are shown in the first column, in which the first four are `Hadoop` stress tests. We use the *Min-max game* strategy and set $\boldsymbol{A}$ to $\{\{a_1\}, \{a_2\}, ..., \{a_n\}\}$ in the experiments. The second column, a_{worst}, is the disturbance action that leads to the worst performance. Columns 3–5 show the optimal performance values for three environments. E_i is the *ideal* environment without any disturbance actions. E_d is the *disturbed* environment injected by action a_{worst}. The environment E_d^{ic} is similar to E_d with action a_{worst}, but it uses the optimal configuration for E_i rather than that for E_d. Note that the performance value is a relative value based on the value of E_i, so the performance value of E_i is always 1. Columns 6 and 7 show the optimal configurations for E_i and E_d, respectively. We use the serial numbers in Tab. 2 to represent the parameter names (e.g., c_1 refers to `number_of_datanode`). Notice that for each workload, we only search for the configuration parameters related to it.

From these results, we make the following observations. First, as expected, the disturbance actions affect the performance of the distributed application, as seen by comparing E_i and E_d^{ic}. An exception is `ZooKeeper`, for which the performance value is almost the same. This is likely because `ZooKeeper` is not compute-intensive like `Hadoop`, and the effect of environment anomalies is less on its main function of maintaining a quorum. Second, a comparison of Columns 6 and 7 shows that the optimal configurations for ideal and disturbed environments are different. Also, different workloads have different optimal configurations. Third, one should consider when to use an optimal configuration for a disturbed environment. By comparing the relative performance values, we can see that although the performance of E_d is a bit worse than that of E_i, it is still an improvement when compared with E_d^{ic} in almost all cases. This means that using the optimal configuration for a disturbed environment, rather than an ideal environment, may be a better choice in practice. The RIOT framework provides a flexible platform to find these configurations.

5 Related Work

The closest related work in performance tuning of distributed systems is by Babu et al. on `StarFish` [5, 6]. `StarFish` is a self-tuning analytics system for `Hadoop`. It uses model-based estimation to predict `Hadoop` job performance, and then uses the prediction to find good settings for configuration parameters. However, `StarFish` only targets `Hadoop` and assumes an ideal environment. Our approach considers the effect of environment anomalies and can be applied to arbitrary distributed systems. Lubke et al. provide an architecture that allows network emulation of standard client/server-based architectures [7]. The main goal is to

get precise and accurate performance measurements, while our goal is to tune the performance. Random fault-injection based methods [11] and test description languages [8] are also proposed for distributed application resiliency testing. One drawback of such application-level fault injection is that the injected faults may not be justifiable in practice. In our approach, the disturbance actions we allow guarantee that every observed failure is justifiable. Our previous work `SETSUDO` [10] targets robustness testing of cloud applications. It allows a tester to specify testing policies using application-dependent abstraction labels that expose internal states of the application. Here, we treat the applications as black boxes without knowing the internal states of the application, and we focus on performance rather than robustness.

6 Conclusions

In this paper, we proposed a black-box approach to tune the performance of distributed services along with consideration of environmental anomalies. We implemented this approach in a framework called RIOT for arbitrary distributed systems. It extends the automatic service orchestration framework Juju to simulate and inject disturbance actions, and performs measurements and other health checks. We performed experiments on some popular distributed service applications, which show that environmental anomalies can significantly affect their performance, and the optimal configurations for ideal environment and disturbed environment are different. We believe practical deployment of distributed services can benefit from considering anomaly-aware configuration search.

References

1. Ganglia. `http://ganglia.sourceforge.net/`.
2. Juju. `https://juju.ubuntu.com/`.
3. Juju Charms. `https://jujucharms.com/`.
4. LXC. `http://linuxcontainers.org/`.
5. S. Babu. Towards automatic optimization of mapreduce programs. In *In SOCC '10*, pages 137–142, 2010.
6. H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *In CIDR '11*, pages 261–272, 2011.
7. R. Lubke, R. Lungwitz, D. Schuster, and A. Schill. Large-scale tests of distributed systems with integrated emulation of advanced network behavior. *WWW/Internet*, 10(2):138–151, 2013.
8. P. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 29(4):11:1–11:38, 2011.
9. I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.
10. G. B. A. G. P. Joshi, M. Ganai and N. Papakonstantinou. Setsudo: Perturbation-based testing framework for scalable distributed systems. In *In TRIOS '13*, 2013.
11. A. Tseitlin. The antifragile organization. *CACM*, 56(8):40–44, 2013.
12. T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. In *In SIGMETRICS '03*, pages 196–205, 2003.